

软件工程最佳实践

Software Engineering Best Practices *Lessons from Successful Projects in the Top Companies*

(美) Capers Jones 著 吴舜贤 杨传辉 韩生亮 译



机械工业出版社
China Machine Press

软件工程最佳实践

Software Engineering Best Practices *Lessons from Successful Projects in the Top Companies*

本书深入地探讨了其他软件工程文献中很少涵盖的如下主题：为什么软件行业开发出了超过2500种之多的编程语言，软件质量传统定义的诸多问题，“代码行数”和“平均缺陷成本”等违反标准经济学假设的通用度量指标的种种缺陷。本书指出，数量巨大的“新”项目实际上只不过是遗留应用的替代品，这说明为寻找那些已遗失的初始需求而进行的数据挖掘应该成为软件项目的标准实践。本书还讨论了如何将机构精简、裁员的危害降至最小等棘手的社会工程问题。本书说明了如何有效地使用成熟、可靠的工程过程来规划、估算、安排进度和管理各种类型的软件项目。

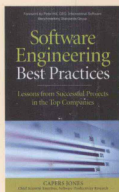
本书主要内容：

- 管理敏捷、层级式、矩阵式和虚拟软件开发团队
- 使用JAD、QFD、TSP、静态分析、审查和其他具有出色成功案例记录的方法优化软件质量
- 使用高速的功能性度量指标评估软件项目的生产力和软件质量水平
- 规划最佳软件组织，从小型团队到1000人以上的大型组织

作者简介

Capers Jones Capers Jones & Associates公司的创始人及CEO，软件生产力研究所（SPR）的创始人和前总裁，兼首席科学家。他还是软件质量世界大会的主题演讲人，是国际功能点用户组（IFPUG）的终身会员，被信息技术软件质量联盟（CISQ）评为“杰出顾问”。Capers的研究涵盖了软件质量评估、质量度量、软件成本与进度估算以及软件度量指标等。

原书封面



McGraw-Hill
全球智慧中文
<http://www.mheducation.com>

Mc
Graw
Hill
Education



客服热线: (010) 88378991 88361066
购书热线: (010) 68326294 88379649 68995259
投稿热线: (010) 88379604

数字阅读: www.hzmedia.com.cn
华章网站: www.hzbook.com
网上购书: www.china-pub.com

上架指导: 计算机/软件工程

ISBN 978-7-111-44540-1



9 787111 445401 >

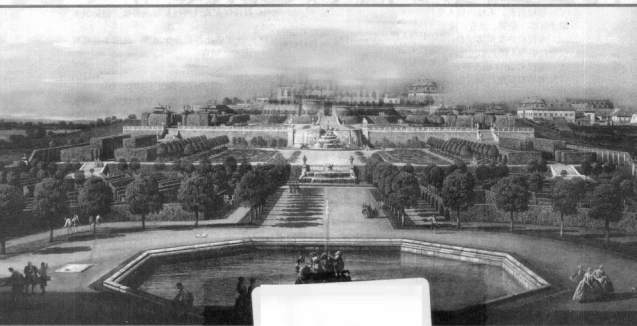
定价: 99.00元

软 件 工 程 技 术 丛 书

软件工程最佳实践

Software Engineering Best Practices *Lessons from Successful Projects*
in the Top Companies

(美) Capers Jones 著 吴舜贤 杨传辉 韩生亮 译



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

软件工程最佳实践 / (美) 琼斯 (Jones, C.) 著; 吴舜贤, 杨传辉, 韩生亮译. —北京: 机械工业出版社, 2013.11
(软件工程技术丛书)

书名原文: Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies

ISBN 978-7-111-44540-1

I. 狄… II. ① 琼… ② 吴… ③ 杨… ④ 韩… III. 软件工程 IV. TP311.5

中国版本图书馆 CIP 数据核字 (2013) 第 252917 号

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问 北京市展达律师事务所

本书版权登记号: 图字: 01-2012-7782

Capers Jones : Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies (978-0-07-162161-8).

Copyright © 2010 by McGraw-Hill Education.

All Rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including without limitation photocopying, recording, taping, or any database, information or retrieval system, without the prior written permission of the publisher.

This authorized Chinese translation edition is jointly published by McGraw-Hill Education (Asia) and China Machine Press. This edition is authorized for sale in the People's Republic of China only, excluding Hong Kong, Macao SAR and Taiwan.

Copyright © 2014 by McGraw-Hill Education (Asia) and China Machine Press.

版权所有。未经许可, 对出版商的任何部分不得以任何方式或途径复制或传播, 包括但不限于复印、录制、录音, 或通过任何数据库、信息或可检索的系统。

本授权中文简体字翻译版由麦格劳-希尔(亚洲)教育出版公司和机械工业出版社合作出版。此版本经授权仅限在中华人民共和国境内(不包括香港特别行政区、澳门特别行政区和台湾)销售。

版权 © 2014 由麦格劳-希尔(亚洲)教育出版公司与机械工业出版社所有。

本书封面贴有 McGraw-Hill Education 公司防伪标签, 无标签者不得销售。

本书从软件工程的宏观层面, 以专业的视角, 摆事实、列数据, 对比各种软件工程实践, 剖析优劣, 洞悉软件工程的成败, 揭露各种软件工程实践的伪真理, 深刻指出软件项目中存在的各种问题的实质, 并给出中肯的改进建议和解决方案。这些最佳实践来自作者所研究的全球超过 600 家知名软件公司和美国 30 余个大型政府机构, 可以称得上是软件行业半个世纪以来全球范围内软件工程实践的精华。本书共分 9 章。第 1 章给出软件工程“最佳实践”的定义, 第 2 章探讨软件工程领域的 50 条最佳实践, 第 3 章展望未来软件开发的状况, 第 4 章评估学习新的软件工程信息的 17 个渠道, 第 5 章展示许多不同类型组织结构的考察结果, 第 6 章讨论涉及项目管理的职能, 第 8 章探讨编程和代码开发工作以及度量编程效率和编程质量的方法等, 第 9 章讨论正式审查、静态分析以及其他 17 种不同形式测试方法的优势和劣势等。

机械工业出版社(北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑: 谢晓芳

北京市荣盛彩色印刷有限公司印刷

2014 年 1 月第 1 版第 1 次印刷

186mm × 240mm × 31.75 印张

标准书号: ISBN 978-7-111-44540-1

定 价: 99.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

购书热线: (010) 68326294 88379649 68995259

投稿热线: (010) 88379604

读者信箱: hzjsj@hzbook.com

译者序

自20世纪40年代中期现代计算机诞生以来,作为现代计算机“灵魂”的计算机软件经历了程序设计、软件设计和软件工程等发展阶段。尤其是自1968年提出“软件工程”概念以来,软件行业经历了风起云涌的飞速发展。各种各样的编程语言、形形色色的开发方法、多如牛毛的软件工具、不计其数的软件应用不断如雨后春笋般涌现。迄今为止,也许还没有哪个工程行业像软件工程领域如此欣欣向荣。这种蓬勃发展的强劲势头的确令人欢欣鼓舞。从表面上看,我们有充足的理由对软件工程的未来充满信心和期待。

对于软件这样一个年近古稀的行业,我们理所当然地认为它应该是非常成熟、可靠的。但事实是这样吗?在软件行业欣欣向荣的背后,我们也看到了令人不安的一面。正如Peter R. Hill (ISBSG CEO)所说的那样,熙熙攘攘的软件产业就像时装行业一样,不断被各种流行时尚裹挟着前行,却始终无法找出解决软件工程中存在的最根本问题的解决办法。屡见不鲜的软件工程项目失败、进度落后、预算超支、软件质量低劣、程序崩溃、数据丢失、客户严重不满甚至法庭上兵戎相见,数十年来似乎从未间断。静下心来,梳理软件工程的纷繁复杂,去芜存菁,细数软件行业的种种实践、方法、工具、语言、组织、专业、度量、质量等方方面面,细细想来,正如本书作者在书中屡屡强调的那样,软件工程还真是一门“手艺”而非一个真正的工程学科。

本书作者Capers Jones是一位睿智的长者。他科学、严谨的研究方法和一丝不苟的态度,使译者对Capers产生了由衷的敬佩!Capers潜心研究软件工程数十年,以深邃的洞察、广博的知识、严谨的思维和严格的论证,洞悉软件工程的是非与成败,揭露各种软件工程实践的伪真理,深刻指出软件项目中存在的各种问题的实质,孜孜不倦地为我们提出各种改进建议和解决方案。Capers的研究,没有空洞的说教,每每以超过15 000个项目的真实状况和历史数据为依据,对比各种方式方法的优劣,让我们能够看清软件行业各种“传闻”的真实面貌。30余年来,作者在他出版的17本书中对各种软件工程实践进行了深刻剖析,对软件行业的种种传闻和说法给出了自己的看法与建议。他的很多观点,多次引起业界热烈讨论,甚至引发了激烈论战。

在这本书里,作者站在软件工程的宏观层面,以专业的视角,摆事实、列数据,对比各种软件工程实践,剖析优劣,并给出中肯的改进建议,所谓“最佳实践”是为如此。这些最佳实践,来自作者所研究的全球超过600家知名软件公司和美国30余个大型政府机构,可以称得上是软件行业半个世纪以来全球范围内软件工程实践的精华。阅读本书,跟随作者将软件工程的各种实践放到显微镜下仔细审视,很多地方常常引起译者的诸多思考和强烈共鸣,不觉为作者的精辟论述拍案叫绝。作者对软件工程的方方面面都进行了探讨,

涵盖面之全、讨论点之细，令译者也无不惊讶。看完本书，译者有一种强烈的冲动，要将这本书和作者对软件工程的深刻见解尽快分享给无数仍然被软件项目中的各种问题所困扰、苦苦挣扎在软件工程实践第一线的广大中国同行们。

如果你是软件公司高管，本书论述的最佳实践将会使你的公司从中受益。如果你是软件项目经理，你将能够更加清楚地看清工作中遇到的各种问题的根本原因。如果你是软件测试人员，本书将使你能够以更高的视角看待软件测试工作，明白软件测试的核心目的不是发现了多少缺陷，而是如何给客户交付高质量的软件。如果你是质量保证人员，你将会更加深刻地明白，我们是来帮助团队提高软件质量而不是来阻碍团队前进的。如果你是软件开发人员，本书将使你跳出编码工作本身，从更高层面深刻认识软件开发所涉及的方方面面，更加明白为什么当前编码工作量不及软件项目全部工作量的 40%。

无论你是谁，本书关于软件职业、专家、组织结构及学习渠道的分析，能使你清晰了解你在软件行业的位置，明了你所从事的工作所需要的专业知识以及如何获取这些专业知识。无论如何，站在软件工程的宏观层面，纵览软件工程的前世今生，品析软件行业的是是非非，探讨软件项目的改进建议，预测软件开发的未来发展，能够使软件工程从一门手艺演变为一门真正的工程学科，将对软件行业大有裨益。

这本书是 *Capers* 的又一鸿篇巨制，英文版厚达 600 余页，因而翻译该书绝非一人之功所能轻易为之。本书的成功翻译是团队协作的结晶。吴怡编辑统筹全书翻译，耐心指导、仔细审阅，为翻译的顺利进行提供了很多宝贵的建议和帮助。杨传辉负责本书第 1~3 章及前言的翻译，韩生亮负责第 7~8 章的翻译，吴舜贤负责第 4~6 章、第 9 章以及前言、致谢等其他部分的翻译。

在本书翻译过程中及完稿之后，很多朋友参与了译稿审校，在这里对他们的帮助深表感谢：王剑华、张晓辉、卢永安、付勇智、叶慧、孟美。很多知名软件人士及朋友对本书某些词汇、句子的翻译进行了深入讨论并给出了翻译建议，在此一并表示感谢：崔启亮、朱少民、王正、徐毅、朱筱丹、刘圣文、赵霞等。最后，感谢家人张少真的全力支持使我能够全身心投入翻译之中。

本书译者均来自软件工程实践第一线，对技术图书的翻译素怀敬畏之心和惶恐之情，唯恐不能精准、地道地表达作者原意以致误导读者，因而在翻译过程中查阅了大量资料，通过微博、邮件等向原书作者 *Capers Jones* 先生以及许多知名软件人士广为求教，对书中的很多术语、段落、表达方式反复推敲、再三讨论以确定最终译法，倾尽全力将翻译错误减至最少。但由于时间紧迫加之水平有限，书中错误疏漏之处在所难免，恳请广大读者不吝赐教、批评斧正。

吴舜贤

序

软件工程是一个关于人的行业——需要并使用软件的人、构建软件的人、测试软件的人、管理软件项目的人以及那些支持和维护软件的人。可是，为什么软件工程的重点仍然集中在技术上了呢？

软件开发业务有多少年的历史了？让我们假定它已有 55 年之久。这意味着软件行业已历经了整整一代人。这些人受到培训，为软件行业奉献了他们毕生的精力，而现在，他们要么已经退休，要么即将退休。在他们的岁月里，他们无数次目睹了“更好方法”的承诺——那些鬼使神差的银弹。现在，软件工程行业已经有了成千上万的新编程语言、形形色色的新方式方法、多如牛毛的新软件工具。有时候，软件行业似乎就像时装行业一样，强劲地被各种各样的流行时尚推动着不断前行。许多从业者崇拜特殊方法、实践或者工具，并成为了虔诚门徒，至少在某一时间段内是这样。但是，当我们收集并分析各种指标数据的时候，我们发现了一个可悲的事实：作为一个行业，我们其实没有取得多大的进步。在按时按预测成本顺利交付高质量软件方面，仍然没有任何重大突破。而怀疑论者也会问：“软件工程是自相矛盾的行业吗？”

我们对技术的执着和眷恋蒙蔽了我们的双眼。我们不希望或者无法看到，我们真正需要的是基础，即良好的工程实践。在这本书里，Capers Jones 主张，如果软件行业要想被认可为一个工程学科而不是一门手艺，就必须采用稳固可靠的工程实践方法经济高效地交付可接受的软件产品。技术令人着迷，但当需要出色地完成工作并交付良好质量的软件产品时，它并不是最重要的因素。人们的工作方式、人们做出的抉择以及人们选择去遵守的规范，都比技术选择对软件工程成功的影响大得多。

一定有很多次，Capers 会感觉自己就像个先知，孤独地行走一片荒漠之中。他一直孜孜不倦地为软件行业提供各种指导和教诲，努力推动软件行业成为一个真正的专业和工程学科。他的努力跨越 28 年之久，成就了 16 本软件专业著作。很多次我都怀疑 Capers 晚上是否睡觉。他的书籍草稿总是源源不断地出现在我的收件箱里，无论是白天还是夜晚。当你读到一些东西，然后对自己说“好吧，这很有道理；真的，这显而易见啊”的时候，你会认识到作者已经做了一件了不起的事情。Capers 就是这样的一位作家。他所写的东西常常引人入胜、易于理解，而且注重实效。我手头上的他的书籍，经常被我一翻两页，还贴满了记事便签。这证实了他所著书籍的实用性。

Capers 有一种能力，使得他可以置身事外，认真观察那些仍然长期困扰软件工程行业的各种问题的实质。Capers 无所畏惧地批判着他认为对软件行业非常危险的那些实践——在这本书里，他的批评目标是“平均缺陷成本”和“代码行”度量指标。毫无疑问，尽管

他废除这些有害度量措施的理由非常充分，但他的观点仍会引起争论。辩论和讨论将会非常激烈，而这对于软件行业来说仍是期待已久的幸事。Capers 站在专业的角度点燃导火索，引发了一场关于这些度量指标到底是否有害的论战。

在这本书中，作者也将软件质量放到了显微镜下仔细审视。他把软件质量描述为软件工程成功的关键因素——比任何其他因素对软件成本、进度和最终成功都具有更大影响的驱动因素。由于 Capers 对一些常见的软件质量定义提出了挑战，因而同样会充满争论。

纵贯全书，作者始终在鲜明强调软件度量措施和度量指标的重要性。Capers 对软件行业缺乏度量实践和使用他描述为“危险”的那些度量指标进行了批评。鉴于使用度量措施和度量指标很少，软件行业理应受到激烈批评和强烈质疑。正如 Capers 所断言的，当我们无法给出统计学上的量化证据来加以证明时，像“最佳实践”这样的术语令人尴尬。

软件工程已有 55 年的悠久历史了，软件工程应该变得成熟了。在本书中，Capers Jones 对软件工程中的人与管理问题的重点强调，为软件工程实现真正的成熟指出了光明大道，并由此软件行业要成为一个真正工程学科的努力方向。

——Peter R. Hill

国际软件基准组织 (ISBSG) 有限责任公司 CEO

前 言

本书写作之时，正值全球经济衰退开始。因此，与过去的一些书相比，在应对软件工程的问题上，本书提供了一个新的研究方向。

由于经济衰退，本书有了许多新的素材，这些素材涉及：裁员和机构精简；美国和其他国家之间不断变化的经济平衡；经济衰退时期的软件经济学等。

当2008年金融危机和经济衰退开始时，软件工程并没有立即受到影响，但随着时间推移，风险投资基金开始逐渐耗尽。软件公司其他形式的融资变得困难起来，因此到2009年中期，软件工程类职位开始受到裁员的影响。鉴于在经济下行期间，诸如质量保证和技术写作等专业职位往往在第一轮就被波及，因此这些职位受到的影响甚至更加严重。

经济衰退带来的一个意想不到的副产品是，由于软件工程师供应的大量增加加上福利待遇的降低，因此这使美国成为软件外包的一个候选国家。

截至2009年，美国、印度以及中国之间的成本差距在减少，相对于国际合同，美国国内合同的便利可能也证明这是对美国软件工程界有益的一件事。

随着经济衰退的持续，高成本的软件正在经历空前考验。经济衰退也突出了这样一个事实，即软件一直都是不安全的。由于经济衰退，诸如有价值信息的盗窃、身份盗用甚至监守自盗等网络犯罪现象迅速增加，“网络钓鱼”或试图使用虚假信息以获取个人银行账户访问权限的现象也在大幅上升。

从笔者的角度来看，经济衰退已经凸显了，要使软件工程成为一门可靠的工程学科，而不是一门手艺，需要改进以下四个重要领域：

1. 需要通过提高应用程序的免疫力水平，以及包括编程语言自身更好的安全特性等措施来逐步改善软件的安全性，而访问控制和权限控制一直也是软件工程的薄弱环节。

2. 需要从缺陷预防方法和缺陷去除方法两方面来改善软件质量。近50年来，不佳的软件质量一直破坏着软件经济，而这种情况不能再持续下去了。每一款重要应用软件都需要有效地结合审查、静态分析和测试等方法来改善软件质量，而只靠软件测试这一种方法并不足以获得高品质的软件。

3. 为了更好地理解软件开发和软件维护的真实经济价值，需要对软件的度量方式加以改进。这意味着需要考虑活动级别的软件成本，也意味着要分析传统度量指标（如“平均缺陷成本”和“代码行”等）的缺陷，这些度量指标违反了标准经济学的有关法则。

4. 由于经济衰退，新的软件开发正不断放缓，因此需要更好地理解软件维护和改造的经济价值。维护和改造遗留应用程序的方法变得越来越重要，为了获取“丢失”的软件需求和业务规则而对遗留应用程序进行挖掘的方法也同样重要。

截至 2009 年,绝大多数企业和软件工程师,对生产力或软件质量都没有一个有效的度量方法,这不符合一个真正的工程学科的标准。有效度量指标的缺乏加上现有危险度量指标的使用意味着,诸如敏捷开发、Rational 统一过程(RUP)以及团队软件过程(TSP)等软件方法的有效性评估将更加困难。

虽然在经济增长时期,度量措施以及软件工程方法和实践的有效性评判能力的缺失只是有些不便,但在经济衰退时期,这却是一个严重问题。糟糕的度量方法使得诸如“最佳实践”等短语在描述软件时非常尴尬,因为大量最佳实践的主张并不是基于有效度量指标而获得的可靠度量结果。

本书尝试说明度量指标与度量措施的结合能证明“最佳实践”的有效性,并且希望为软件工程建立一个良好的经济基础。本书所述的“最佳实践”是指那些由量化数据至少提供了一种临时能力以能够判断其有效性的软件实践。

这本书共分 9 章,每一章都涉及了一套技术和非技术问题。

第 1 章:软件最佳实践的介绍和定义

由于许多软件应用程序在第一次交付后,可能会持续使用 25 年甚至更长时间,所以软件工程不能仅仅关注软件开发活动。软件工程需要考虑软件交付多年后的维护和改进工作。软件工程还需要包括为提取或“挖掘”遗留应用程序而使用的有效方法,以恢复丢失的业务规则和软件需求。

负责软件维护的软件工程师数量要比负责新软件开发的工程师多得多。许多软件工程师的任务是维护并非自己开发的应用程序,这些软件可能是用“过时”的编程语言来编码的,并且代码本身既没有任何功能说明也没有有效的注释。

软件工程“最佳实践”不是一个“放之四海而皆准”的技术。最佳实践的评估要求那些实践在拥有 100 个功能点或者更少功能点的小型应用程序、拥有 1000 个功能点的中型应用程序以及可能拥有超过 10 万个功能点的大型系统等规模的软件应用项目中进行评价。

此外,那些对 Web 应用软件和 IT 应用软件有效的“最佳实践”并不一定要对那些嵌入式应用、系统软件以及武器系统软件具有同等良好的效果。

由于在应用规模和应用类型上变化巨大,本书对这两个方面的最佳实践均做了评估。

第 2 章:50 个软件最佳实践概述

该章探讨了 50 个软件工程最佳实践。并非所有的实践都是纯技术性的。例如,在经济衰退时期,如果处理不当,企业裁员将会损坏其多年的经营效益。

该章涉及软件开发的最佳实践、软件维护的最佳实践、项目管理的最佳实践以及社会学的最佳实践,例如,那些涉及裁员的实践办法,这方面经常会处理不当,例如淘汰经理和主管人员过少,但淘汰软件工程师和软件专家过多。

除裁员以外的其他方法，例如，缩短所有员工的工作时间以及减少福利待遇，往往是首选办法，这也相当于裁员。

其他最佳实践领域包括安全控制、质量控制、风险分析、软件治理、软件开发、软件维护以及遗留应用改造等。

该章部分内容曾经应用于软件诉讼，这些诉讼中的某些软件项目实践未能符合软件工程最佳实践成为原告提起诉讼索赔的原因之一。

第3章：2049年的软件开发和维护预览

因为软件工程最佳实践是在接近2009年时才开始实施的，所以我们很难想象它的变化和改进。第3章让我们提前穿越到2049年，并探讨那个时候的软件工程将是什么样的。那时的世界可能是这样的，我们所有人都通过社交网络来相互联系，软件工程的工作可以很容易地分配给分别位于许多不同国家的软件工程师。

该章还展望了具体技术的发展，如数据挖掘技术在需求收集方面的作用，以及可获取大量认证的可重用材料。可能也包括可用于重要议题上对信息进行积累和分析的智能工具与搜索机器人。

鉴于技术的高速发展，可以预见的是，到2049年，计算设备、网络和通信渠道等硬件将极其成熟。但软件工程往往滞后于硬件的发展。为了跟上硬件和网络的发展速度，软件工程需要在安全性、质量以及可重用性上做出重大改进。

第4章：软件人员如何学习新技能

由于经济衰退，纸质图书和软件杂志出版商正面临着严重的打击，其中很多出版商都在裁员。在线出版和电子书籍，如亚马逊的Kindle和索尼的PR-S05，正在迅速壮大。就提供者和用户而言，Web出版物、博客以及Twitter也在不断扩张。

第4章评估了17个学习渠道，即传播和学习新软件工程信息的渠道。在学习效率和成本效益方面，对每个渠道都进行了排名。针对每种渠道的未来也做了长期预测。

已评估的某些学习渠道包括：传统的纸质图书、电子图书、软件期刊、电子期刊和博客、wiki站点、商业培训、在职培训、学术教育、现场技术大会、网络研讨会以及在线会议等。就成本效益而言，电子媒体已经超越了印刷媒体，而在学习有效性方面也正挑战着传统媒体。

第4章还为软件工程师、软件质量保证人员、软件测试人员、软件项目办公室人员以及软件经理推荐了一些课程。虽然当今的教学课程足以满足主流软件工程的需要，但在诸如规模估算、评估、规划、安全、质量控制、维护、革新以及软件工程经济分析等主题上，仍然缺乏可靠的教育。

软件度量指标几乎不被学术界关注，而对诸如平均缺陷成本和代码行等常见指标的缺

陷也很少有批判性的分析。

虽然功能性度量指标在很多大学有所教授，但是很少有人针对行为异常或者违反制造业经济原则的旧度量指标进行经济分析。特别是，固定成本对生产力的影响没有涉及，这也是代码行和平均缺陷成本为什么对经济分析无效的主要原因。

第5章：软件团队的组织 and 专业化

软件工程组织的范围很广，从生产小型应用程序的一个人独立公司，到超过 1000 人的大型组织（可能是雇用超过 5 万软件从业人员的公司的一部分）均包括在内。

除了软件工程师本身，大型软件工程组织雇用了超过 90 种各类专家，例如质量保证专家、技术文档作家、数据库管理员、安全专家、网站管理员以及度量指标专家等。

第 5 章展示了许多不同类型组织结构的考察结果，其中包括结对编程、小型敏捷开发团队、层级式组织、矩阵式组织以及在地理上分散的虚拟组织等。该章同时还说明了组织和管理诸如软件质量保证、测试、技术文档以及项目管理办公室等专家最行之有效的方

式。例如，对于大公司里的大型软件项目，独立的维护团队和独立的测试团队往往比开发团队自身进行软件维护与测试更为有效。软件专家和通才必须共同努力，而组织结构对整体业绩有着很强的影响。

第6章：项目管理和软件工程

众所周知，许多软件项目的规模估算是不正确的，因此提交的项目进度日程对开发团队的能力来说可能太短。项目管理中的这些失误可能会影响软件项目，使之要么完全失败，要么出现严重的成本超支和工期延误。

第 6 章涉及了一些关键管理职能，例如项目大小估算、规划、评估、进度跟踪、资源跟踪、基准以及变更管理等，如果处理不好这些管理职能，就会导致软件工程的失败。

对于每个稍大的软件项目，第 6 章建议我们收集那些可以作为项目基线 and 基准的软件质量与生产率方面的数据。对生产力和质量数据的收集应该是普遍的，而不是罕见的例外情况。

使用严格的度量方法并获取度量结果是职业化的象征。而未能很好地进行度量则说明软件工程还不是一门真正的工程学科。

第7章：需求、业务分析、架构及设计

在编写任何代码之前，有必要先理解用户需求。这些需求应映射到有效的软件架构，然后再翻译成详细设计。此外，新的应用程序应该置于整体企业应用项目组合的背景下加以评估。在 20 多种形式的需求方法和 40 多种类型的设计方法中，软件工程师可以有很多

种选择。

该章探讨了处理需求和设计问题使用最广泛的方法，并展示了最适合它们的应用程序类别和类型。敏捷方法、统一建模语言（UML）以及很多其他技术都将在这里加以讨论。

大概在 2009 年，大型企业的全部投资项目组合可能拥有超过 5000 个应用程序，总计超过 10 万个功能点。项目组合可以包括内部应用程序、外包应用程序、商业应用程序以及开源应用程序等。

项目组合还可以包括 Web 应用程序、信息技术应用软件、嵌入式软件以及系统软件等。由于经济衰退，公司高管们是否了解企业项目组合的规模、内容、价值、安全等级以及质量水平等，变得越来越重要。

过去，难以量化各种应用程序的规模，这已经阻碍了项目组合分析。而今天，在商业应用程序、开源应用程序以及内部应用程序上，新型高速规模估算方法已经开始消除这些历史问题。现在，在短短的几天到几个星期时间里，已经能够估算成千上万个应用程序的规模了。

第 8 章：编程和代码开发

该章以一个非同寻常的观点来探讨编程和代码开发工作。截至 2009 年，大约有 2500 种编程语言及其派生语言。该章提出了“软件工程为什么有如此之多的编程语言”的问题。

第 8 章还提出了“过多的编程语言对软件工程专业到底是有益还是有害”这样的问题。此外，还讨论了许多应用程序使用 2 ~ 15 种不同编程语言的原因。总体结论是，虽然有些编程语言对软件开发有益，但是 2500 种语言的存在对软件维护工作来说简直就是一场噩梦。

该章还建议美国国家编程翻译中心记录所有已知语言的语法，并协助将那些用已过时语言编写的应用程序转换成现代编程语言编写的应用程序。

该章还包括，在源代码中发现各种类型的 bug 信息，以及与功能测试和回归测试等公开活动相比，由软件工程师自己执行的最有效的“个人”缺陷预防和缺陷去除方法。

例如手工检查和单元测试等个人方法经常很难度量。然而，志愿者通过“私人”的缺陷去除活动来发现缺陷并做信息记录，因此这些数据常常可用。

该章还介绍了度量编程效率和编程质量水平的方法。由于挑战传统的在经济上无效的“代码行”（LOC）度量指标，该章内容备受争议。LOC 度量指标对高级编程语言极为不利，同时也误导了经济分析。

即便是在 2009 年，代码行度量指标不能处理需求、设计、界面或文档等工作，而这些非编码活动的费用占了软件开发项目总费用的 60% 以上。

另一种可供选择的是功能性度量指标，它可以处理所有已知的软件工程活动。然而，软件功能性度量指标的计算速度缓慢并且成本昂贵。大约 2009 年，开始出现了新型的高速

功能性度量指标，并且这些指标的使用也在不断扩大。

第9章：软件质量：软件工程成功的关键

长期以来，在与软件工程相关的各个技术环节中，软件质量一直是最薄弱的环节之一。该章试图涵盖所有影响软件质量的主要因素，其中包括缺陷预防方法和缺陷去除方法。

该章探讨了正式审查、静态分析以及其他17种不同形式测试方法的优势和劣势。此外，该章还涉及了各种令人烦恼的度量指标，这些度量指标降低了人们对软件质量的理解。例如，比较流行的“平均缺陷成本”指标实际上对软件质量不利，使漏洞百出的应用软件也能达到最低的平均缺陷成本！此外，软件质量具有的经济价值远远超过了单纯的缺陷去除成本的经济价值，而这种价值不能使用平均缺陷成本指标来体现。

该章的主题是，软件质量是一个在软件成本、进度以及项目成功上比其他指标具有更大影响力的因素。但是，糟糕的度量方法使得难以开展有效的软件工程经济研究。

该章饱受争议的原因是，我们质疑了两种常见的质量定义。质量意味着“符合需求”这一质量定义被质疑的理由是，许多需求是有害或者“有毒的”，并且不应该实现。质量意味着满足一系列以“ility”结尾的特性（比如“可移植性”（portability）），这一质量定义受到挑战的理由是这些特性中的某些既不能预测也无法度量。人们需要一个这样的质量定义，“质量”既可以在应用程序开始运行之前预测，又可以在应用程序完成之后度量。

质量是软件工程成功的关键。但在打开通向专业化的大门之前，我们有必要了解如何度量软件质量以及软件经济性。该章的结论是，无法度量自己成果的活动不是一个真正的工程学科。现在非常需要研究诸如潜在缺陷和缺陷去除效率水平等关键议题。

截至2009年，大多数软件项目都包含过多的错误或缺陷，而在软件交付之前去除的缺陷或错误少于全部缺陷的85%。每一名软件工程师和软件项目经理都应该知道需要什么样的审查、静态分析以及阶段测试结合，以实现缺陷去除效率水平到达99%。如果没有基于精确度量的缺陷预防和缺陷去除知识，软件工程就是不当用词，而软件开发也只是一门手艺而非一种真正的职业。

软件工程最佳实践的总体目标 本书的灵感之一来自于1982年的一本老书，这本书是《The Social Transformation of American Medicine》（《美国医学的社会转型》），作者是普利策奖得主Paul Starr（保罗·斯塔尔）。

在我读保罗·斯塔尔的书之前，我并不知道，原来150年前，医学院的学生通过两年的学习后，就可以被授予医学学位，并且没有任何医院实习或高级住院专科实习的要求。其实，大多数在培训的医生从来没有进入过医院或者医学院。更令人吃惊的是，学生在进入医学院的时候并没有提供任何大学学位或者高中文凭证明。在美国，超过50%的医生从来没有上过大学。

保罗·斯塔尔的书详细介绍了美国医学协会的尝试，以改善医生的学术培训、建立职

业过失的准则以淘汰庸医、提高医生的职业地位等。在保罗·斯塔尔的书中，很多经验对软件工程很有价值。

这本关于软件工程最佳实践的书的主要目标是，为促使软件工程建立在软件质量和生产力精确度量这一事实基础之上而提供激励措施。

随着经济衰退的持续，软件行业越来越需要减少软件项目失败、加快软件交付、降低软件维护费用。没有软件工具、开发方法、编程语言以及软件组织结构等的有效性的精确度量，这些要求都无法实现。

精确的度量是获得更好软件质量和安全性的关键。而更好的软件质量和更高的安全性是让软件工程变成一种真正职业的关键，即等同于那些已经取得了成功的旧工程领域。

软件工程的度量结果也会产生更多、更好的软件基准，而这反过来将为已证明是有效的软件工程方法提供有力的证据。本书的整体主题是，作为软件工程成功的先驱条件，软件工程需要更完善的度量措施、更全面的软件基准、更严格的质量控制以及更高的安全性。

致谢

一如既往地首先感谢我的妻子——Eileen，是她的支持使我在过去的28年里完成了16本著作。

还要感谢那些为本书及笔者以前的很多书籍提供真知灼见的很多同行们：Michael Bragen 和 Doug Brindley，我任职的前一家公司——软件生产力研究所（SPR）的CTO和总裁；Tom Cagley，国际功能点用户组（IFPUG）总裁；Bob Charrette，ITABHI公司总裁；Coverity Systems公司的Ben Chelf；微软公司的Steven Cohen；James Madison大学的Taz Dougherty博士；Chas Douglass，软件生产力研究所前总裁；Larry Driben博士，Pearl Street软件公司总裁；Gary Gack，Process Fusion公司总裁；Jonathan Glazer，PowerBees公司总裁；Scott Goldfarb，质量与生产力管理集团总裁；Steve Heffner，Pennington Systems公司CEO；Peter Hill，国际软件基准组织（ISBSG）CEO；软件工程研究所（SEI）的Watts Humphrey；Ken Hamer-Hodges，Sipantic公司总裁；IBM Rochester研究中心的Steve Kan博士；北德克萨斯大学的Leon Kappleman博士；Ravindra Karanam，Unisys公司软件业务部门的CTO；Dov Levy，Dovel Systems公司总裁；Tom Love博士，Shoulders公司总裁；Steve McConnell，Construx公司总裁；Michael Milutis，信息技术指标与生产力研究所（ITMPI）主管；Peter Mollins，Relativity Technology公司首席营销官；Unisys公司的Prasanna Moses；Walker Royce博士，IBM Rational事业部负责人；Akira Sakakibara博士，IBM东京研究中心的杰出科学家；Tony Salvaggio Computer Aid Inc.（CAI）总裁；Paul Strassmann，信息经济出版社总裁（美国国防部前CIO）；以及Cem Tanyel，Unisys应用软件开发服务部门的副总裁兼总经理。

我们还应当向两位在软件专业化领域做出大量卓越贡献的高管表示特别的赞誉。已故

的 James H. Frame, IBM 圣特雷莎实验室主管及随后的 ITT 康涅狄格州斯特拉特福德编程开发中心副总裁。已故的 Ted Climis, IBM 副总裁, 他在 IBM 识别了大量的关键软件角色。上述两位都清楚地认识到软件对于公司商业运营至关重要, 需要以最高卓越标准来设计和构建软件产品。

谨以此书献给那些潜身研究并推动软件工程领域不断进步的同行们! 其中一些人包括: Allan Albrecht、Barry Boehm、Fred Brooks、Tom DeMarco、Jim Frame (已故)、Peter Hill、Watts Humphrey、Steve Kan、Steve McConnell、Roger Pressman、Tony Salvaggio、Paul Strassmann、Jerry Weinberg 以及 Ed Yourdon。

目 录

译者序

序

前言

第 1 章 软件最佳实践的介绍和定义 1

1.1 什么是“最佳实践”？如何进行评估	5
1.2 软件开发、部署以及维护的多种路径	7
1.3 软件部署的路径	9
1.4 维护和部署的路径	10
1.5 软件开发、部署以及维护的量化	12
1.6 软件工程中的关键主题	14
1.7 方法、实践以及社会学因素的总排名	18
1.8 总结	28
参考文献	28

第 2 章 50 个软件最佳实践概述 31

2.1 最大限度地减少裁员所带来的危害	33
2.2 技术人员的积极性和动力	35
2.3 经理和高管的积极性与动力	37
2.4 软件人才的选拔和招聘	39
2.5 软件人员的考核以及职业生涯规划	39
2.6 软件应用早期的范围控制	40
2.7 软件应用的外包	41
2.8 使用承包商和管理顾问	44
2.9 选择软件方法、工具以及做法的最佳实践	45
2.10 认证方法、工具以及实践	49
2.11 软件应用的需求	54
2.12 用户参与软件项目	55
2.13 软件应用中的行政管理支持	56
2.14 软件架构和设计	57

2.15	软件项目规划	58
2.16	软件项目的成本估算	59
2.17	软件项目的风险分析	61
2.18	软件项目的价值分析	63
2.19	取消或拯救陷入困境的项目	64
2.20	软件项目的组织结构	65
2.21	培训软件项目经理	67
2.22	培训软件技术人员	69
2.23	使用软件专家	69
2.24	软件工程师、专家以及管理人员的认证	71
2.25	软件项目中的沟通	73
2.26	软件的可重用性	74
2.27	可重用材料的认证	76
2.28	编程	80
2.29	软件项目管理	82
2.30	软件项目的度量和指标	82
2.31	软件的基准和基线	84
2.32	软件项目的里程碑和成本跟踪	86
2.33	软件发布前的变更控制	87
2.34	配置控制	89
2.35	软件质量保证	90
2.36	审查以及静态分析	92
2.37	测试和测试库的控制	95
2.38	软件的安全性分析与控制	98
2.39	软件的性能分析	100
2.40	软件的国际标准	101
2.41	软件中的知识产权保护	101
2.42	防止病毒、间谍软件以及黑客	103
2.43	软件的部署和定制	114
2.44	培训软件应用的客户或用户	115
2.45	软件应用部署后的客户支持	116
2.46	软件担保和召回	117
2.47	软件发布后的变更管理	118
2.48	软件的维护和功能增强	119
2.49	软件应用的更新和发布	121

2.50 遗留应用的终止或撤销	122
2.51 总结	123
参考文献	123
第 3 章 2049 年的软件开发和维护预览	133
3.1 引言	133
3.2 需求分析	134
3.3 设计	136
3.4 软件开发	138
3.5 用户文档	140
3.6 客户支持	140
3.7 部署和客户培训	142
3.8 软件维护和功能增强	143
3.9 软件外包	146
3.10 软件包评估和收购	152
3.11 技术选择和技术转型	154
3.12 企业架构和项目组合分析	156
3.13 软件学习预览	158
3.14 尽职调查	160
3.15 认证和授权	162
3.16 软件诉讼	164
3.17 总结	166
参考文献	167
第 4 章 软件人员如何学习新技能	168
4.1 引言	168
4.2 软件学习渠道的演变	169
4.3 软件工程师当前需要学习哪些技术主题	171
4.4 软件工程专家	173
4.5 软件专业的种类	175
4.6 专家与普通软件人员的大概比率	178
4.7 评估软件工程师所使用的学习渠道	179
4.8 需要额外教育的软件领域	196
4.9 软件学习的新动向	197
4.10 总结	198

4.11 软件管理和技术类主题课程	198
参考文献	201
第5章 软件团队的组织 and 专业化	203
5.1 引言	203
5.2 量化组织结果	204
5.3 割裂的信息技术和系统软件世界	204
5.4 集中办公与分布式开发	205
5.5 软件专家组织面临的挑战	207
5.6 由小到大的软件组织结构	209
5.7 大型公司的专家组织	226
5.8 总结	254
参考文献	255
第6章 项目管理和软件工程	257
6.1 引言	257
6.2 软件规模估算	263
6.3 软件进度与问题跟踪	296
6.4 软件基准	300
6.5 总结	318
参考文献	318
第7章 需求、业务分析、架构及设计	322
7.1 引言	322
7.2 软件需求	323
7.3 软件需求方法论及实践	337
7.4 业务分析	347
7.5 软件架构	349
7.6 企业架构师	352
7.7 软件设计	356
7.8 总结	360
参考文献	361
第8章 编程和代码开发	364
8.1 引言	364

8.2 编程语言开发简史	364
8.3 我们为什么会有超过 2500 种编程语言	366
8.4 编程语言普及性的探索	369
8.5 我们到底需要多少种编程语言	372
8.6 建立一个国家级的编程语言翻译中心	374
8.7 为什么大多数软件都使用 2 ~ 15 种编程语言	377
8.8 有多少程序员使用多种编程语言	378
8.9 源代码中通常会出现何种类型的缺陷	380
8.10 软件缺陷的逻辑和属性	382
8.11 软件源代码缺陷的预防和去除	387
8.12 编程缺陷预防方法	388
8.13 缺陷去除方法	396
8.14 “代码行”度量方法的经济学问题	403
8.15 总结	415
参考文献	416
第 9 章 软件质量：软件工程成功的关键	419
9.1 引言	419
9.2 软件质量定义	421
9.3 软件质量度量	441
9.4 软件缺陷预防	453
9.5 软件缺陷去除	462
9.6 软件质量专家	467
9.7 软件质量的经济价值	479
9.8 总结	486
参考文献	486

软件最佳实践的介绍和定义

2008 年 9 月 15 日, 当开始写这本书的时候, 21 世纪最严重的经济危机伴随着 Lehman 兄弟申请破产保护而突然降临。迄今为止, 所有的证据都表明, 一场痛苦且漫长的经济衰退将会持续一年甚至更长的时间。尽管在 2009 年中期, 部分企业有恢复的迹象, 但是失业的人数、丧失抵押品赎回权的人数以及破产的人数却在持续增长。即使最乐观的经济恢复预测也指向了 2010 年年底, 而悲观的预测甚至指向了 2011 年或 2012 年。事实上, 这次经济衰退可能会造成金融行业永久性的变化, 并且, 目前尚不清楚, 失业者将会在什么时候返岗。在失业率排名前 10% 的一些州, 美国的经济状况是不容乐观的。

在经济日益衰退的情况下, 软件行业也无法幸免。多数软件公司将会倒闭, 并且由于许多公司正在缩小规模, 同时尽力节省开支, 因此数以千计的员工将会下岗。

从以往看, 软件成本已经成为企业开支的重要组成部分。软件成本很难控制, 并且质量低劣、安全边际以及其他长期存在的问题一直严重影响着软件成本。

不景气的软件工程行业给人们带来了有严重缺陷的经济模式, 并且也导致了经济的衰退。随着经济衰退的加深, 我们迫切需要认真审视和软件工程相关的基本问题, 比如质量、安全、结果的度量以及最佳开发实践等。本书将讨论以下重要议题(重大经济危机时期):

- ❑ 最大限度地减小由裁员和缩小规模带来的损失。
- ❑ 优化软件质量控制。
- ❑ 优化软件安全控制。
- ❑ 从定制开发到注册组件的迁移。
- ❑ 通过开发新软件来代替改造遗留软件。
- ❑ 度量软件的经济价值和风险。
- ❑ 通过规划和评估以减少意外超支。

软件工程过去一直是一门手艺或者艺术形式, 而不是一个真正的工程领域。这本书并没有提供灵丹妙药, 但是如果“软件工程”想要成为一个真正的职业术语, 那么这本书就讨论了一些在软件工程中需要改进的重要技术领域。

只要软件应用是通过手工编码来进行构建的, “软件工程”这个词就是名不副实的。从个性化的开发转变成注册组件的构建有非常广阔的前景, 同时这样做也会使软件工程学科和软件成本结构有空前的巨变。

在 2009 年, 已经有十几本优秀的软件工程书籍出版, 因此有些读者可能会问我们为什

么还需要一本关于软件工程的书呢。我想最主要的原因就是，我们需要考虑大型软件应用的主要成本驱动因素。然而，截至 2009 年，这种状况一直是令人揪心的。

在软件诉讼中，笔者曾作为专家证人。在工作的过程中，笔者抽查了 600 多家企业和政府机构的软件工程结果，通过分析这些结果，笔者有了一个惊人的发现，那就是我们的软件项目在发现错误和已取消项目上的花费超过了其他的任何一个方面。表 1-1 罗列了，截至 2009 年，软件行业里 15 个主要的成本驱动因素（按照降序）。

表 1-1 2009 年应用程序的主要成本驱动

1	发现和修复错误	9	项目管理
2	已取消的项目	10	改造和迁移
3	制作英文词源 ^①	11	创新（新型软件）
4	安全漏洞和攻击	12	项目的失败和软件灾难的申诉
5	需求变更	13	培训和学习软件
6	编程或编码	14	避免安全漏洞
7	客户支持	15	组装可重用组件
8	会议和沟通		

对于一个真正的工程领域来说，这 15 个主要的成本驱动因素本来不应该是这个样子的。在理想情况下，我们应该在创新和编程上投入更多的资金，而在修复 bug、已取消项目以及各种类型的问题上投入较少的资金，如修复安全漏洞等。在真正的工程领域，我们也应该能够使用比目前数量多得多的零缺陷可重用组件。

本书的主旨是将精益求精的软件工程和最佳实践放置在一个健全的量化基础之上。如果软件工程变成一门真正的工程学科，并且成功的项目比失败的项目多，那么成本驱动因素将会改变。本书旨在帮助改变软件成本驱动因素，并且希望在十年内实现该目标，最终让它们遵循如表 1-2 所示的模式。

表 1-2 2049 年修订后的成本驱动列表

1	创新（新型软件）	9	需求变更
2	改造和迁移	10	制作英文词源
3	客户支持	11	编程或编码
4	组装可重用组件	12	发现和修复漏洞
5	会议和沟通	13	安全漏洞和攻击
6	避免安全漏洞	14	已取消的项目
7	培训和学习软件	15	项目失败和灾难的诉讼
8	项目管理		

根据该修订的成本驱动表，缺陷修复、软件失败以及已取消项目的成本已经从最高降到了最低。在软件开发中，只要有更好的安全控制，安全攻击恢复的成本也将会降到最低。

^① “制作英文词源”是指和大型软件项目相关的 90 个文档。许多大型应用在创建文档的时候，花费的时间和资金比在编写源代码上花费的多。——译者注

修订后的列表将是新型软件的创新。编码仅仅排在了第 11 的位置, 因为与 2009 年相比, 到那个时候, 一门真正的工程学科就能够运用更多的零缺陷可重用组件了。如果软件工程想要变成一门真正的职业, 而不是一门仅仅使用边缘方法的手艺, 并且这些边缘的方法经常导致失败, 那么修订后的软件成本驱动列表就展示了这种开叉模式的庐山真面目。

即使在不远的将来, 软件工程变成了一门真正的工程学科, 而不再是一门手艺, 但是由于软件行业已经有 60 年的历史了, 因此遗留软件的更新、迁移以及维护的成本依旧会是所有成本中最高的。在每个拥有 50 多年历史的行业里, 维护和改进一直是总成本中不可轻视的两个方面。

本书还有另外一个目的, 那就是探讨软件应用整个生命周期中的最佳实践, 从早期的需求到部署, 再到后来的维护等。由于一些大型应用使用的年限超过了 30 年, 因此本书涵盖的主题非常广泛。本书不仅涉及开发的最佳实践, 还涉及部署、维护、改造的最佳实践, 除此之外, 甚至还包括软件在最终结束使用时, 如何退役应用程序的最佳实践。

由于许多大型项目最终以失败告终, 甚至都还没有交付, 因此本书会谈到许多有意思的最佳实践, 这些最佳实践试图回过头去拯救那些夭折的项目。如果该项目已经不再拥有什么价值, 因而拯救该项目是不可行的, 那么本书也会提供一些与终止缺陷应用对应的最佳实践。

当前, 对遗留软件来说, 在维护和改进上投入的人力超过了在开发上投入的人力, 然而在软件工程的相关文献中, 却很少提及维护和改进方面。

尽管有许多与软件工程相关的优秀书籍, 但是为了给软件形式的改进和创新提供资源, 我们仍然需要改善软件的质量控制和安全控制。当然, 我们也需要在遗留软件的维护和改进上投入更多的精力。

截至 2009 年, 在软件行业里投入了超过 50% 的资金在软件的缺陷修复、安全缺陷或者灾难上, 如已取消的项目等, 而在实际的创新和新形式软件上的投入不到 10%。

如果我们能够使开发实践、质量实践以及安全实践专业化, 那么我们就有希望使灾难、错误修复以及安全修复的费用降低到 15% 以下。如果真能如愿, 那么我们可以腾出来 40% 的资金, 并且将这些资金投入新型软件中。

从需求到最后交付, 拥有 10 000 个功能点(信息系统提供的一种业务功能的度量单位)的应用程序在每个功能点上的花费大约为 2000 美元。而在这 2000 美元的花费中, 用于查找和修复错误的花费“遥遥领先”, 竟然超过了 800 美元。像这种拥有 10 000 个功能点的应用程序, 需要用 48 ~ 60 个月才能完全交付。因此开发这种应用程序, 总成本就相当高, 并且成本分布的不合理导致了最终相当失败的工程实践。

如果我们能够开发超过 10 000 个功能点的应用程序, 并且每个功能点的花费少于 500 美元, 而在查找和修复错误上花费少于 100 美元, 那么通过更好的缺陷预防方法和运用零缺陷的可重用组件, 我们将会极大地改善软件经济地位。对于拥有 10 000 功能点的应用程序, 开发时间在 12 ~ 18 个月之间也是很有价值的, 因为对于不断变化的市场行情, 较短的时间能够促使更快的反应。这些目标在理论上可能运用了最先进的软件方法和实践。但是从理论走向实践, 就需要在质量控制上做出重要的转变, 并且也要从手工编码迁移到

基于零缺陷标准组件的构建上来。一个悬而未解的问题是，这种转变是否能在十年之内完成。现在还不能肯定十年的时间是否充足，但是可以肯定的是，这样深刻的转变至少需要十年。

截至 2009 年，几乎所有的大型项目都超出了预算，通常延迟交付，并且在最终交付的时候，应用程序还包含许多 bug。更糟糕的是，在功能点超过 10 000 的大规模项目中，约有 35% 的项目会夭折。

由于已取消项目的花费比成功项目的花费更高，因此与已取消的大型软件应用相关的浪费也是巨大的。对于功能点在 10 000 左右的大型项目，成功的项目在每个功能点上的花费大约是 2000 美元；而被取消的项目，由于经常延迟交付并且超支，因此在每个功能点上的花费可能达到了 2300 美元。

在所有的工程行业里面，软件行业拥有最高的失败率。对于任何一个行业，如果项目的延迟交付率多达 75%，并且大型应用程序的取消率多达 35%，那么这个行业就不能是一门真正的工程学科。

对于拥有 10 000 个功能点的软件应用，一旦部署完毕，并交付给用户使用，在每个功能点上的维护费用大约是 200 ~ 400 美元/年。在这些维护费用中，约有 50% 是用来修复错误的，另外的 50% 是用来改进和添加新的功能的。

即使上面的成本分配，也是需要改进的。在理想情况下，缺陷修复的成本应该降低到每年每功能点 25 美元。使用维护工作台和修复工具应该可以将维护成本降低到每年每功能点 75 美元。应用程序维护和改善的薄弱环节是客户支持，而这些方面依旧是高度劳动密集性的，并且也是普遍欠佳的。

在笔者作为专家证人的诉讼中，笔者在证词和证言中发现，许多软件项目最终通过法院去裁决，原因是已取消或严重超支的软件项目没有遵循良好的工程实践。有 5 个常见的问题伴随已取消或者灾难性的项目：

- 项目启动之前，估算不准确并且过于乐观。
- 在项目期间，质量控制不到位。
- 在项目期间，变更控制不健全。
- 在开发过程中，进度跟踪严重不到位，甚至是误导性的。
- 当问题在第一次出现的时候，并没有快速有效地解决该问题，而是将该问题忽略或者隐瞒。

当成功的项目在经过审查，并完成交付后，成功的项目和失败的项目之间的差距就变得明显了。成功的软件项目一般会有良好的规划和评估、质量控制、变更管理、进度跟踪，并且也善于解决问题而不是忽略问题。除此之外，成功的软件项目倾向于遵循良好的工程实践，而失败的项目一般则不这样做。

证词与法庭的证言透露了更加微妙和深层的问题。截至 2009 年，越来越多的量化数据能够提供有力的证据：某些方法和活动是有价值的，而其他一些方法则是有害的。例如，当进度变缓或者超期时，管理人员经常会采取不明智的做法，如通过绕过审查或者缩短测试时间等行为来赶上工程进度。但是这种做法往往会事与愿违，并且还会使问题变得更糟。为什么软件的项目经理不懂得一个道理呢？那就是有效的质量控制能缩短工期，而粗糙的

质量控制则延长工期。

出现这些错误的原因之一就是，尽管许多关于软件工程和软件质量的书籍都告诉我们该如何有效地进行控制质量，但是它们并没有提供量化的结果。换句话说，软件工程界不需要“怎么做”的信息，而是需要“运用这种方法会导致什么样后果”的信息。例如，下面的信息将会很有用。

“通过对 50 个包含 10 000 功能点的项目样本进行调查，我们发现，在这些项目的开发进度计划中，大约有 36 个月是在进行设计和代码审查，并且最终实现了 96% 的缺陷去除效率。”

“通过对 125 个 10 000 功能点的相似项目样本进行调查，我们发现，这些项目都没有使用设计和代码审查。在这些项目中，有 50 个项目由于未完成而被取消，而剩余的 75 个项目的平均工期都在 60 个月，并且最终的缺陷去除效率仅有 83%。”

对软件开发方法和开发途径的结果进行量化是很有必要的，如敏捷开发（Agile development）模型、瀑布开发（Waterfall development）模型、六西格玛（Six Sigma^①）模型、CMMI（Capability Maturity Model Integrated，能力成熟度模型集成）模型、RUP（Rational Unified Process，Rational 统一过程）模型、TSP（Team Software Process，团队软件过程）模型等。本书尝试为许多常见的开发方法提供量化信息。但是，请注意，混合的方法也很常见，例如，TSP 和 CMMI 结合使用。接下来，我们将会探讨常见的混合模式，但是在处理这些混合模式时也会有很多的变化。

1.1 什么是“最佳实践”？如何进行评估

一本名为“软件工程最佳实践”的书首先应该准确定义“最佳实践”的含义，然后再解释数据的来源，并且包含每个实践的使用背景。一本关于“最佳实践”的书也应该提供量化数据，以展示最佳实践结果。

由于实践方法随着应用程序的规模和类型而变化，因此评估应用程序也就变得困难起来了。例如，对拥有 2500 个功能点的项目，使用敏捷开发是相当有效的；但是对超过 10 000 个功能点的项目，敏捷开发将会迅速失效。对于拥有 10 000 个功能点的项目，目前尚且没有人尝试过使用敏捷开发；也许对于这种规模的项目，敏捷开发本来就是有害的。

为了处理这种情况，一种包括规模和类型的近似评分方法已经开发出来了。我们可以使用如表 1-3 所示的标准对方法进行评分，评分的范围为 -10 分到 10 分。另外，这个表也包含了生产力和质量的近似影响。这个评分方法可以适用于特定的功能点范围，例如，拥有 1000 或者 10 000 个功能点；也可以适用于特殊类型的软件，如信息系统、Web 应用程序、商业软件以及军事软件等。

① Six Sigma（中文译名“六西格玛”）是一种管理方法。在整个企业流程中，Six Sigma 是指每百万个机会当中有多少缺陷或失误，这些缺陷或失误包括产品本身以及产品生产的流程、包装、运输、交货期、系统故障、不可抗力等。——译者注

改进的中点度量法或者“平均”度量法都是传统的方法，例如，运用瀑布模型进行开发的组织，要么不使用软件工程研究所的 CMM（能力成熟度模型），要么就处于级别 1。截至 2009 年，这种相对原始的组合仍然或多或少地被广泛应用在开发方法中。

我们应该理解一个重要的主题，那就是应该快速地将质量提升到一个比生产力更高的级别，接着再来改善生产力。这样做的原因是，总体而言，发现和修复 bug 是软件开发中花销最大的活动。质量主导生产力，没有上乘的质量，想要提高生产力，简直是白日做梦。

软件工程行业一直有一个悬而未解的严重问题，那就是度量的实践方法屈指可数，以至于量化的结果也十分匮乏。许多收费的工具、语言以及方法都声称自己应该成为一个最佳实践，然而，在质量和生产力的上，它们关于实际成效方面的经验数据却少得可怜。

本书试图另辟蹊径，尝试一种与众不同的新方法。一种语言、工具或者方法要想成为最佳实践，就需要与前 15% 的软件项目关联，并且这些项目是经过作者本人以及同事度量和研究的。一种具体的方法或工具要包含在最佳实践的集合中，就必须通过量化数据来证明，并且这些量化数据改善了项目进度、工作量、成本、质量、客户满意度或者以上这些因素的组合。此外，只有拥有了足够的数据，才能使用表 1-3 所示的评分方法。

这个标准提出了三个要点。

要点 1： 软件应用有规模不等的多个数量级。对于拥有 1000 个功能点的小型项目，有些方法可能称为最佳实践；但是对于拥有 10 000 个功能点的大型系统，这些方法可能就不是同样有效了。因此本书中的评分方法使用规模作为一个标准，用来衡量“同类最佳”的状态。

要点 2： 软件工程不是一个“一刀切”的行业。软件有许多不同的类型，例如，嵌入式应用程序、商业软件包、信息技术项目、游戏、军事应用、外包应用及开源应用等，同时不同类型的软件应用不一定使用相同的语言、工具或者开发方法。因此对每种应用软件，本书只考虑能产生最佳结果的途径。

要点 3： 对于所有的活动，工具、语言和方法并不是同样的有效或重要，例如，功能强大的编程语言 Objective C 在编码速度和代码质量上，明显会产生有益的影响。但是，这种编程语言运用在需求蔓延、用户文档或者项目管理上却没有任何效果。因此，短语“最佳

表 1-3 软件开发方法与实践的评分标准

分数	生产力的改进	质量的改进
10	25%	35%
9	20%	30%
8	17%	25%
7	15%	20%
6	12%	17%
5	10%	15%
4	7%	10%
3	3%	5%
2	1%	2%
1	0%	0%
0	0%	0%
-1	0%	0%
-2	-1%	-2%
-3	-3%	-5%
-4	-7%	-10%
-5	-10%	-15%
-6	-12%	-17%
-7	-15%	-20%
-8	-17%	-25%
-9	-20%	-30%
-10	-25%	-35%

实践”也不得不甄别哪些具体的活动是可以改进的。由于活动包括开发、部署以及部署后的维护和改进，因此甄别“最佳实践”相当复杂。事实上，对于大型应用程序，开发时间可能长达5年，安装可能长达1年，而最终的使用时间可能长达25年。在超过30年的时间里，可能有数以百计的事情发生。

上述因素的结果是，为软件工程选择一组最佳实践将是一个相当艰巨的任务。我们需要根据应用的规模、类型以及活动来评估每种方法、工具或者语言的有效性。本书将探讨在各种情况下的最佳实践：

- 按应用程序规模分类的最佳实践。
- 按软件类型（嵌入式、Web、军事等）分类的最佳实践。
- 按活动（开发、部署、维护）分类的最佳实践。

在2009年，由于不具备国家级认证、许可授权、广泛实验、正式专业以及拥有已被证明是行之有效的技术和方法的可靠经验事实体系，软件工程还不是一门真正的工程职业。当然，有许多国际标准。此外，各种不同类型的认证可能是建立在自愿的基础上。所以，迄今为止，既没有哪个标准也没有哪个认证能够证明它确实能够改善软件项目的成功率。

这并不是说，认证和标准没有任何价值，而是说通过量化质量和生产率来证明自己的价值是一项艰巨的任务。笔者观察发现，与未通过认证的测试人员相比，在相似的应用程序上，个别的测试认证似乎产生了更高水平的缺陷去除效率。在进行计数实验时，实验结果表明，认证的功能点计数人员会比未通过认证的计数人员产生更准确的结果。然而，我们需要更多更好的数据，来作为一个令人信服的例子，以证明认证的价值。

至于标准，其结果往往是含糊不清的。例如，没有任何可靠的数据表明，遵循ISO质量标准后，会获得较低水平的潜在缺陷或者较高水平的缺陷去除效率。许多安全标准看似展示其在减少安全漏洞数量方面的改进，但是这些数据是稀疏的，并且是未经对照研究核实的。

1.2 软件开发、部署以及维护的多种路径

本书其中的一个目的就是，展示一套可以被奉为圭臬的“路径”。该路径包括：从软件项目正式启动，到软件开发，再到最后的成功交付等。在交付之后，该路径将继续指导后续的维护和改进。

由于许多路径都是基于应用程序的规模和类型的，因此可能存在一条路径网络。软件工程成功的关键是找到一条特定的路径，并且这条路径对特定项目会产生最佳结果。有些路径会包括敏捷开发或包括TSP（团队软件过程），有些路径则会包括RUP，而少数的则可能包含传统的瀑布开发方法。

无论使用哪种特殊的路径，为了确保应用程序取得圆满的成功，我们必须包含一些基本的目标。

- 项目规划和估算必须是优秀并且准确的。
- 质量控制必须是严格的。
- 变更控制必须是妥善的。

- 进度和成本跟踪必须是及时的。
- 结果的度量必须是巧妙并且准确的。

典型的开发路径如图 1-1 所示，该图展示了应用在三种不同规模范围的软件应用上的开发方法和质量实践。

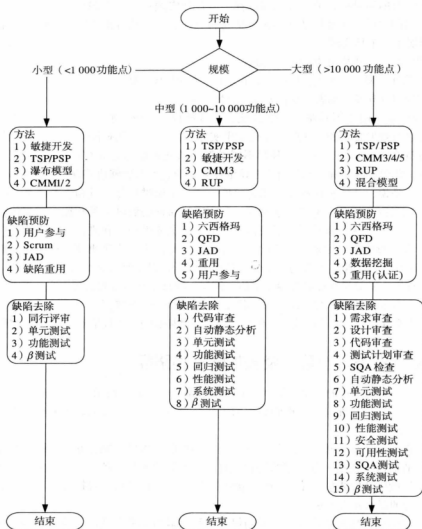


图 1-1 根据应用程序规模分类的开发实践

首先，需要稍微地解释图 1-1 所示的路径，方法框顶部的方法表明，该方法有最高的成功率。例如，对于少于 1000 个功能点的应用，敏捷开发绝对是首屈一指的；但是，对于

大一点的应用程序，TSP 和 PSP 则更加有效。然而，所有框中的方法已经成功地运用于所示规模的应用程序上了。

沿着箭头的方向向下移动，缺陷预防和缺陷去除框展示了审查、检查以及测试的最佳组合。正如你所看到的，与拥有少于 1000 功能点的小型应用程序相比，较大的应用显得更加复杂，同时也需要多种类型的缺陷去除方法。

继续以路径作为类比，我们会发现有很多路径可能导致延期和失败，只有屈指可数的路径会获得成功的结果，并且这些结果还伴随着高品质、短的开发周期以及低成本等。实际上，穿越一个大型项目的路径就像穿越一个迷宫一样。大多数的路径将是死胡同，而检查度量和量化数据就好像站在天梯上俯视整个迷宫一样，这样我们就可以看清楚哪些路径会导致最终的成功，同时也可以避免走一些不必要的弯路。

1.3 软件部署的路径

最佳实践并不是只限于开发。在文献中，一个主要的分歧就是安装和部署大型应用的最佳实践。像那些只使用诸如 Windows Vista、苹果 OS X、微软 Office 以及 Intuit Quicken 等个人电脑软件的读者，可能会觉得奇怪为什么部署竟然是一件重要的事情。对于许多应用来说，通过下载、CD 或者 DVD 来安装可能只需要短短的几分钟。实际上，对于软件即服务（Software as a Service, SaaS）来说，例如，谷歌的文字处理和电子表格等应用，我们甚至不需要下载就可以使用，而这些应用都是运行在谷歌的服务器上的，并且从来不在用户电脑上运行。

然而，对于大型主机应用（如电话交换系统）、大型主机操作系统以及 ERP（Enterprise Resource Planning，企业资源规划）等应用安装包，部署和安装可能需要一年甚至更久。这是因为这些应用不仅需要安装，而且为了符合本地的业务和技术需求，甚至还需要大量的定制。

此外，培训大型应用的用户也是一项重要而费时的活动，我们可能需要安排一系列的课程，并且花费几个星期来给用户上课。另外，还需要为用户、维护人员、客户支持人员以及其他客户创建大量的自定义配套文档。很少有文献会涉及大型应用安装的最佳实践，但是我们有必要考虑这些最佳实践。

不仅软件开发的路径重要，软件交付给用户的路径也同样重要，并且在随后软件的使用过程中，软件的维护和改善路径也相当重要。图 1-2 展示了软件在三种不同情形下的典型安装途径，这些软件包括：云计算技术中的软件即服务（SaaS）软件、自助的软件以及需要顾问和安装专家的软件。

SaaS 是不需要安装的。对于自安装的软件，无论从网上下载还是通过 CD 或者 DVD 物理安装都是很常见的，并且也很容易完成。然而，偶尔会遇到一些问题，例如，诺顿杀毒软件，除非我们先卸载旧版本，否则新版本是没法安装的。但是诺顿杀毒软件的旧版本是如此复杂，以至于 Windows 上一般的卸载程序没法卸载诺顿杀毒软件。最终，赛门铁克（Symantec）提供了专门的卸载工具，因此我们在安装诺顿杀毒软件的新版本之前，必须首先安装赛门铁克的卸载工具。

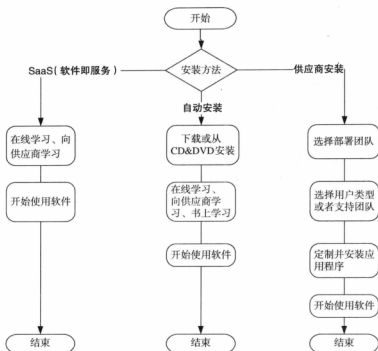


图 1-2 不同类型的部署实践

然而，真正复杂的安装程序是与大型主机应用相关的程序，并且需要定制和安装。例如，某些像 ERP 一样复杂的大型应用安装包，有时甚至需要由 25 名顾问以及 25 个内部人员组成的安装团队花一年的工夫才能完成安装。

由于使用这些大型应用的用户涵盖了几十种不同的组织，例如，会计、市场营销、客户支持、生产等，因此我们需要创建各种自定义用户手册和客户培训课程，才能满足客户的需求。

大型成套软件从交付之日算起，直到它们开通，并且不同的用户开始大规模使用，大约会花费一年的时间。毫无疑问，大型应用程序的安装、部署以及用户的培训不是一件微不足道的事业。

1.4 维护和部署的路径

一旦软件应用安装并且投入使用，随着时间的推移，以下几种类型的变更将会出现。

- ❑ 所有的软件应用都有 bug 或缺陷，并且当我们发现这些 bug 或缺陷时，我们应该修复它们。
- ❑ 随着业务的发展，新特性和新需求将会不断出现，因此我必须不断更新已经存在的应用，才能满足当前用户的需求。

- 当政府的授权或者新法律出现变化时，例如，税收结构变化，应用程序必须能及时响应，有时通知的时间会很短。
- 随着软件使用年限的增加，软件架构的落后可能会使软件的性能降低，并且出现新的 bug 和缺陷。因此，如果该软件一直有商业价值，我们也许有必要“改造”该遗留应用。改造包括以下主题：通过重组或重构以降低复杂度，识别并清除容易出错的模块，也包括在做以上工作的同时增加一些功能。改造是一种特殊形式的维护，同时各种文献也应该更多地涉及维护。
- 经过多年的使用之后，日益陈旧的遗留应用也许会超出它们的使用年限，并且需要替换。然而，再开发一个已存在的应用程序和开发一个新的应用程序不太相同。由于原始的需求和功能说明滞后并且不符合当前的形势，因此可以利用数据挖掘技术从代码中提取已有的业务规则。

因此这本书尝试说明的最佳路径不仅包括开发，还包括部署、维护以及改进。图 1-3 说明了在维护期间，我们应该遵循的三种较为常见却重要的路径。

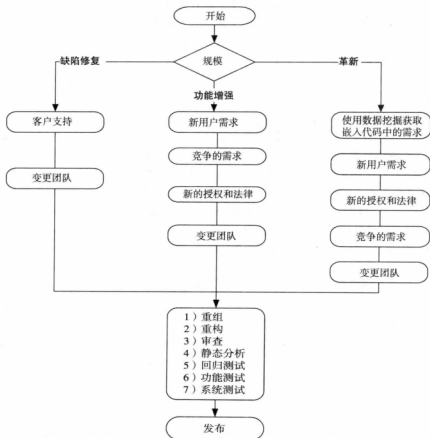


图 1-3 软件维护和功能增强的主要形式

正如在图 1-3 所看到的,软件维护不是一个“一刀切”的改进形式。遗憾的是,与软件开发的文献相比,软件维护的文献寥寥无几。软件的缺陷修复、改进以及革新是迥然不同的活动,并且需要不同的技能,有时可能需要不同的工具。

开发拥有 10 000 ~ 100 000 功能点规模的大型应用是一项浩大的工程,至少需要 5 年的时间。部署这样的大型应用程序需要花费 6 ~ 12 个月的时间。一旦安装,大型应用可能会使用至少 25 年。在使用期间,维护和缺陷的修复将是持续不断的。在某些时候,为了降低复杂性,并且可能为了迁移到新的文件结构或新的编程语言,改造或修复应用的情况可能会经常发生。因此,最佳实践的分析需要持续至少 30 年的时间。开发成本仅仅占总成本中的一小部分。这本书高瞻远瞩,并且尽量涵盖整个项目周期中的所有最佳实践,这个项目的周期包括从软件应用开始开发的第一天,直到 30 多年后最后一个用户停止使用该软件。

1.5 软件开发、部署以及维护的量化

本书将包括生产率基准、质量基准以及一些有效的数据,这些数据来自大量工具、方法和编程实践。另外,本书也包括基于培训成本和部署方法的量化数据。这些数据有几个出处,而最大的出处便是作者自己,因为作者本人在 1973 ~ 2009 年期间,曾对数百个客户进行过深入的研究。

其他关键数据来源有两个:一是收集了美国 SPR (Software Productivity Research, 软件生产力研究所) 有限责任公司的基准,二是收集了非营利性的 ISBSG (International Software Benchmarking Standards Group, 软件基准组织) 的数据。除此之外,选定的数据将会从其他渠道引入。这些数据来源于 David Consulting Group、Quality/Productivity 咨询公司以及 Longstreet 咨询公司的 David Longstreet。其他最佳实践的信息来源包括,在软件工程领域里,当前关于软件工程和各种门户网站的文献,例如,由 ITMPI (Information Technology Metrics and Productivity Institute, 信息技术度量和生产力研究所) 提供的最优秀的门户网站。当然也会包括来自 SEI (Software Engineering Institute, 软件工程研究所) 的信息。也会引用其他的专业协会的信息,例如 PMI (Project Management Institute, 项目管理协会) 和 ASQ (American Society for Quality, 美国质量协会),尽管它们并不公布很多量化的数据。

所有这些数据源提供的基准数据主要使用了 IFPUG (International Function Point Users Group, 国际功能点用户组) 定义的功能点。这本书使用 IFPUG 功能点来量化所有数据,这些数据主要用于质量和生产率方面的处理。

功能点也有其他的几种形式,例如, COSMIC (Common Software Measurement International Consortium) 功能点和芬兰功能点 (Finnish function point)。虽然本书不会详细讨论这些可选指标中的数据,但是会讨论基准数据源的引用。本书还会提及其他的指标,例如,用例点、故事点以及目标问题指标等,当然,也会提供参考文献。

(通过使用“代码行”度量指标或者“平均缺陷成本”度量指标来提供准确的基准是不可能的。正如我们将在稍后展示的,这些常见的指标违反了标准经济学的假设,并且也扭

曲了历史数据，以至于掩盖了真正的趋势。)

当然，与最佳实践相反的便是最糟糕的实践。在一些违反合同的诉讼中，笔者曾作为专家证人参与其中，发现证词和审判文件揭示了重大失误的原因，而这些重大失误便构成了最糟实践。为了展示最佳实践与最糟实践之间的差异，本书将不断探讨这些情况。

在最佳实践和最糟实践之间的集合也有许多称为“中性实践”的方法与实践。这些方法和实践也许对某些类型的应用程序有益，然而，对其他应用程序可能略有坏处。但是不管在哪种情况下，使用这些方法后，在生产力和质量上都会造成巨大的差异。

这本书试图通过仔细地度量结果中的经验数据来代替纸上谈兵。当软件行业能够持续、准确地度量业绩时，能够拥有良好的精度来估算项目结果时，能够构建大型应用而无需延期和超支时，能够实现卓越的产品质量并且达到客户满意时，我们就可以堂堂正正地称自己为“软件工程师”了，而非名不副实。只有当我们的成功远远超过了我们的失败时，软件工程才能成为一个严格规范的工程职业。

然而，软件工程另外一个重要缺陷就是普遍缺乏度量措施，许多软件工程的度量既没有确保效率也没有确保质量。在尝试度量时，许多项目使用的度量指标和度量方法存在严重的缺陷。例如，在软件工程界，最常用的指标是使用超过 50 年的代码行 (LOC) 度量。正如第 6 章将要讨论的，LOC 指标并不适合高级语言，并且根本不能度量非编码活动。在笔者看来，经济研究中使用的代码行度量方法造成了专业过失^①。

另外一个缺陷指标是用于度量质量的平均缺陷成本指标。这个指标本质上就不适合评估质量，并且对错误最多的应用，仅能达到我们要求的最低水平。平均缺陷成本不能用来度量零缺陷的应用。如果用于经济研究，笔者认为平均缺陷成本会造成专业过失。

平均缺陷成本指标附带的数学问题会促成一个不成文的说法，那就是“开发中的一个 bug，在交付后需要花费 100 倍的时间去修复。”这个说法并不是建立在时间和运动学之上的，而是由于平均缺陷成本随着缺陷数量的降低而会升高。在部署前后，修复 bug 花费的时间相当。在修复错误上，两种方式花费的时间都在 15 分钟到 8 小时之间。然而，修复一些细微的错误可能需要更长的时间，但是在部署前后它们所需时间基本相同。

无论代码行还是平均缺陷成本都不能用来作为经济分析，或者展示软件最佳实践。因此，本书将使用功能点指标来进行经济研究和最佳实践的分析。如前所述，IFPUG 定义了功能点所使用的具体形式。

还有其他一些正在使用的功能点，例如，COSMIC 功能点、用例点、故事点、Web 对象点、Mark II 功能点、芬兰功能点、特征点以及其他 35 种功能点的变体等。然而，截至 2008 年，在全球范围内，只有 IFPUG 功能点拥有足够多的历史数据，并且能够用来进行经济和最佳实践的分析。芬兰功能点虽然有几千个项目，但是其中大部分来自芬兰，并且它们的工作实践与美国的有所不同。截至 2009 年，许多国家都在使用 COSMIC 功能点，虽然这种情形正在不断地改善，但是 COSMIC 仍然缺乏相当数量的基准数据。

① 专业过失 (professional malpractice) 指专业技术人员在实施其技能时，由于缺少应有的技术或经验，而给他人造成人身伤害或财产损失。这些技术人员主要包括软件工程师、医生、会计师、律师等。——译者注

本书将会提供一些建议的转换规则,这种转换是在其他指标和 IFPUG 功能点之间的,但是本书仍将使用 IFPUG 功能点 4.2 版本中的计数规则来表达实际的数据。

在撰写本书的时候(2008 年年底到 2009 年年初),功能点界已经开始讨论细分功能点,并且为技术工作使用独立的指标,这些技术工作就是把软件安装到不同平台,或者使软件工作在不同的操作系统上。本书也会讨论与质量相关的工作所使用的一个独立指标,例如,审查、测试、可移植性、可靠性等。在笔者看来,在计数实践方面可能的改变看起来像是在掩盖有用的信息,而不是在揭示有用的信息。第 6 章将会详细讨论这些度量问题。

IFPUG 功能点指标并不完美,但是它们为经济分析和最佳实践的识别提供了许多有利的条件。功能点符合标准经济学的假设。它们可以度量信息技术软件、嵌入式应用、商业软件以及其他类型的软件。IFPUG 功能点既可以用来度量非编码活动,也可以用来度量编码活动。在需求和设计阶段,功能点可以用来度量需求缺陷,也可以用来度量代码缺陷。在开发和维护期间,功能点可以用来处理每个活动。除此之外,来自 20 000 多个工程的基准数据适合使用 IFPUG 功能点。在稳定性和灵活性上,没有任何指标能和功能点度量媲美。

一个关键的事实应该是显而易见的,但是遗憾的是,它并非如此。为了展示较高水平的质量和较高水平的生产率,并且为了确定最佳实践,我们有必要适时地拥有准确的度量方式。在过去 50 多年里,软件工程界使用的度量方法和指标都存在严重的缺陷。一个不能准确度量自己绩效的行业是没有资格称作工程学科的。因此,本书的另外一个目的就是,展示如何将经济分析应用到软件工程项目中。本书将会展示以高精度度量生产率和质量的方法。

1.6 软件工程中的关键主题

截至 2009 年,已证明关于软件工程的几个要点是毫无疑问的。成功的软件工程使用先进的质量控制方法、变更控制方法以及项目管理方法。没有严格的质量控制,几乎不可能有成功的输出;没有妥善的变更控制,需求蔓延将会导致意外的延期和超支;没有卓越的项目管理,估算将会不准确,计划将会有缺陷,并且项目跟踪也会出现严重问题,而这些严重的问题可能导致彻底的失败或重大的超支。质量控制、变更控制以及项目管理是三个事关成败的重要主题。本书讨论的最佳实践形式包括以下几种类型。

1. 软件工程的介绍、定义以及排名

定义和排名:

- ☐ 最佳实践
- ☐ 非常好的实践
- ☐ 好的实践
- ☐ 稍微好的实践
- ☐ 中等的实践
- ☐ 差的实践
- ☐ 最差实践

专业过失的定义

2. 50 个最佳实践概述

社交和员工士气的最佳实践概述

最佳实践概述:

- ☐ 软件组织
- ☐ 软件开发
- ☐ 软件质量和安全
- ☐ 软件部署
- ☐ 软件维护

3. 2049 年软件开发和维护预览

2049 年前后的需求分析

2049 年的设计

2049 年的软件开发

2049 年的用户文档

2049 年的客户支持

2049 年的部署和培训

2049 年的软件外包

2049 年的技术选择和技术转让

2049 年的软件包评估和采购

2049 年的企业架构和项目组合分析

2049 年的尽职审查

2049 年的软件诉讼

4. 软件人才如何学习新技术

软件学习渠道的演化

软件专业化的种类

软件学习渠道的评估 (以降序排列):

- 1) 网页浏览
- 2) 网络研讨会、播客以及电子学习
- 3) 电子书籍 (电子书)
- 4) 内部培训
- 5) 使用 CD 和 DVD 自学
- 6) 商业培训
- 7) 供应商培训
- 8) 现场会议
- 9) Wiki 站点
- 10) 模拟网站

- 11) 软件期刊
- 12) 使用书和培训资料自学
- 13) 在职培训
- 14) 导师指导
- 15) 专业书籍、专刊以及技术报告
- 16) 大学本科教育
- 17) 研究生教育

5. 团队组织与专业化

大型团队和小型团队

寻找最佳的组织结构

矩阵组织与层级组织的对比

使用项目办公室

专家与通才

结对编程

为本地开发使用 Scrum 会议

为分布式开发提供通信

内部开发、外包开发或者二者兼而有之

6. 项目管理

度量与指标

评估应用的规模

应用的风险分析

规划与估算

应用软件治理

成本和进度的跟踪

基准与行业规范的对比

通过设定基线来决定过程的改进

已取消的项目和灾难恢复

在外包协议中，最大限度地减少诉讼的可能性

7. 架构、业务分析、需求和设计

软件和业务需求一致

为新应用收集需求

挖掘遗留应用中的需求

需求变更或需求蔓延

需求波动或细微变更

架构在软件中的职责

软件设计方法

需求变更和多版本发布

8. 代码开发

开发方法的选择

编程语言的选择

在同一个应用程序中使用不同的语言

编码技术

代码重用

代码变更控制

9. 质量控制、审查和测试

软件六西格玛

缺陷评估

缺陷和质量度量

设计和代码审查

静态分析

自动测试

配置管理

10. 安全、病毒防护、间谍软件和黑客

安全威胁的预防方法

主动安全威胁的防御

安全攻击的恢复

11. 大型应用程序的部署和定制

选择部署团队

定制大型复杂的应用

开发个性化的培训材料

在开通新应用的同时，运行旧应用

12. 维护和改进

维护（缺陷修复）

功能增强（新特性）

强制变更（政府规定）

客户支持

改造遗留应用

外包维护

13. 采用最佳实践的公司

Advanced Bionics 公司

安泰保险

亚马逊

苹果电脑

Computer Aid 公司

Coverity 公司

Dovel 技术

谷歌

IBM

微软

Relativity Technologies 公司

Shoulders 公司

Unisys

当然，这些主题并不是通向康庄大道的唯一路径。但是，这些主题是问题的核心，这些问题能最终主宰“软件工程”这个术语的命运，能够让软件工程从一个矛盾的修辞转变成一个有效的职业描述，再到最后足够的成熟，而使其他旧形式的工程刮目相看。

1.7 方法、实践以及社会学因素的总排名

任何实践、方法或工具要想成为最佳实践、最佳方法或最佳工具，我们就必须为这些实践、方法或工具提供一些量化的证据，并证明这些实践、工具或方法在质量改进、生产率改进、可维护性改进或者其他一些有形的因素上，确实能够提供价值。

尽管有 200 多个主题会对软件的质量产生影响，但是本书只展示了其中的 200 个。在这 200 个主题中，有 50 个拥有可靠的经验数据，而剩余的则是道听途说或者是不一致的。已收集到的数据来自于 600 家公司的大约 13 000 个项目。然而，这些数据的跨度超过 20 年，因此这些数据可能前后不一致。以下情况很容易出现，有些最佳实践在列表中是不合适的，而改变地方后，就会有更多的可用数据。即使如此，通过对几十或数百个项目进行研究，我们已经证明排名前 50 的方法和实践是有益的，而排名靠后的 50 个方法和实践则是有害的。

在列表中，“好”与“差”之间是大量可能有益也可能偶尔有害的实践。中间的一些实践称为中等实践，它们有时会有微小的帮助，有时则不会，但是不管在哪种情况下，它们似乎都有很大的影响。

尽管本书会按照规模和类型来看待实践和方法，但是它会展示一系列完整的因素，这些因素将以降序排列，而在列表顶部的因素往往是最有说服力的。表 1-4 罗列了对软件应用和项目有影响的 200 个方法、实践以及社交问题。

回想一下，对于少于 1000 个功能点到超过 10 000 个功能点的应用程序，列表中的评分是具体分数汇总的结果。在整张表中，对于系统和嵌入式应用、商业应用、信息技术、Web 应用以及其他类型的应用也分别对它们进行评分。表 1-4 显示了整体的平均分。

表 1-4 软件方法、实践以及结果的评估

方法、实践、结果		平均
最佳实践		
1	可重用性 (>85% 零缺陷材料)	9.65
2	潜在缺陷 < 3.00/ 功能点	9.35
3	缺陷去除效率 > 95%	9.32
4	个体软件过程 (PSP)	9.25
5	团队软件过程 (TSP)	9.18
6	自动化静态分析	9.17
7	审查 (代码)	9.15
8	缺陷去除效率的度量	9.08
9	混合型 (CMM + TSP / PSP + 其他)	9.06
10	可重用功能的认证	9.00
11	可重用功能的变更控制	9.00
12	可重用功能的召回方法	9.00
13	可重用特性的授权	9.00
14	可重用源代码 (零缺陷)	9.00
很好的实践		
15	提前评估潜在缺陷	8.83
16	面向对象 (OO) 开发方法	8.83
17	自动化安全测试	8.58
18	不良修复的注入度量	8.50
19	可重用的测试用例 (零缺陷)	8.50
20	正规的安全分析	8.43
21	敏捷开发	8.41
22	审查 (需求)	8.40
23	时间盒 [⊖]	8.38
24	基于项目活动的生产力度量	8.33
25	可重用设计 (可扩展)	8.33
26	正规的风险管理	8.27
27	自动化缺陷跟踪工具	8.17
28	缺陷起源度量	8.17
29	行业数据基准	8.15
30	功能点分析 (高速)	8.15
31	正规的进度报告 (周报)	8.06
32	正式的度量方案	8.00
33	可重用架构 (可扩展)	8.00
34	审查 (设计)	7.94
35	精益六西格玛	7.94
36	软件六西格玛	7.94
37	自动化成本估算工具	7.92
38	自动化维护工作台	7.90
39	正规的成本跟踪报告	7.89
40	正规的测试计划	7.81
41	自动化单元测试	7.75
42	自动化规模估算工具 (功能点)	7.73

⊖ 一种短迭代方法。——译者注

(续)

	方法、实践、结果	平均
43	Scrum 会议(每日)	7.70
44	自动化配置控制	7.69
45	可重用需求(可扩展)	7.67
46	自动化项目管理工具	7.63
47	正规的需求分析	7.63
48	通过数据挖掘来提取业务规则	7.60
49	功能点分析(模式匹配)	7.58
50	高级语言(当前)	7.53
51	自动化质量和风险预测	7.53
52	可重用的教程资料	7.50
53	功能点分析(IFPUG)	7.37
54	需求变更的度量	7.37
55	针对大型应用程序的正式架构	7.36
56	项目启动之前的最佳实践分析	7.33
57	可重用功能的目录	7.33
58	质量功能展开(QFD)	7.32
59	关键技能的专家	7.29
60	联合应用设计(JAD)	7.27
61	自动化测试覆盖率分析	7.23
62	需求变更的重新估算	7.17
63	度量缺陷的严重级别	7.13
64	正式 SQA 团队	7.10
65	审查(测试材料)	7.04
66	自动化需求分析	7.00
67	DMAIC(设计、度量、分析、改进、控制)	7.00
68	可重用的构建计划	7.00
69	可重用的帮助信息	7.00
70	可重用的测试脚本	7.00
好的实践		
71	Rational 统一过程(RUP)	6.98
72	自动部署支持	6.87
73	自动化程序循环复杂度分析	6.83
74	已取消项目的取证分析	6.83
75	可重用的参考手册	6.83
76	自动化文档工具	6.79
77	能力成熟度模型集成(CMMI 5 级)	6.79
78	年度培训(技术人员)	6.67
79	度量指标转换(自动)	6.67
80	变更审查委员会	6.62
81	正规治理	6.58
82	自动化测试库控制	6.50
83	正规的范围管理	6.50
84	年度培训(经理)	6.33
85	仪表板式的状态报告	6.33
86	极限编程(XP)	6.28
87	面向服务架构(SOA)	6.26

(续)

	方法、实践、结果	平均
88	自动化需求跟踪	6.25
89	总拥有成本 (TCO) 度量	6.18
90	自动化性能分析	6.17
91	过程改进基线	6.17
92	用例	6.17
93	自动化测试用例生成	6.00
94	用户满意度调查	6.00
95	正规的项目办公室	5.88
96	自动建模 / 模拟	5.83
97	认证 (六西格玛)	5.83
98	外包 (维护 \geq CMMI 3 级)	5.83
99	能力成熟度模型集成 (CMMI 4 级)	5.79
100	认证 (软件质量保证)	5.67
101	外包 (开发 \geq CMM 3)	5.67
102	价值分析 (无形价值)	5.67
103	根本原因分析	5.50
104	学习总成本 (TCL) 度量	5.50
105	质量成本 (COQ)	5.42
106	用户参与团队工作	5.33
107	正规的结构化设计	5.17
108	能力成熟度模型集成 (CMMI 3 级)	5.06
109	挣值度量	5.00
110	统一建模语言 (UML)	5.00
111	价值分析 (有形价值)	5.00
效果一般的实践		
112	正规的维护活动	4.54
113	快速应用开发 (RAD)	4.54
114	认证 (功能点)	4.50
115	功能点分析 (芬兰)	4.50
116	功能点分析 (荷兰)	4.50
117	部分代码检查	4.42
118	自动重构	4.33
119	功能点分析 (COSMIC)	4.33
120	部分设计的检查	4.33
121	团队 Wiki 通信	4.33
122	功能点分析 (未经调整)	4.33
123	功能点 (0.001 ~ 10)	4.17
124	自动产生每天的进度报告	4.08
125	用户故事	3.83
126	外包 (海外 \geq CMM 3)	3.67
127	目标问题度量指标	3.5
128	认证 (项目经理)	3.33
129	重构	3.33
130	用户手册的制作	3.17
131	能力成熟度模型集成 (CMMI 2 级)	3.00
132	认证 (测试人员)	2.83

(续)

	方法、实践、结果	平均
133	结对编程	2.83
134	净室开发	2.5
135	正规的设计语言	2.5
136	ISO 质量标准	2.00
中等实践		
137	功能点分析(逆火)	1.83
138	用例点	1.67
139	普通客户支持	1.5
140	部分治理(低风险项目)	1.00
141	面向对象度量指标	0.33
142	手工测试	0.17
143	外包(开发 < CMM 3)	0.17
144	故事点	0.17
145	低级语言(当前)	0.00
146	外包(维护 < CMM 3)	0.00
147	瀑布式的开发	-0.33
148	手工变更控制	-0.50
149	手工测试库控制	-0.50
150	可重用性(平均质量材料)	-0.67
151	能力成熟度模型集成(CMMI 1 级)	-1.50
152	非正式的进度跟踪	-1.50
153	外包(海外 < CMM 3)	-1.67
不安全的实践		
154	测试库控制不足	-2.00
155	使用通才而不是专家	-2.50
156	手工成本估算方法	-2.50
157	生产力度量不准确	-2.67
158	平均缺陷成本度量指标	-2.83
159	客户支持不足	-2.83
160	项目干系人和团队之间的摩擦	-3.50
161	非正式的需求收集	-3.67
162	代码行指标(逻辑 LOC)	-4.00
163	治理不足	-4.17
164	代码行度量指标(物理 LOC)	-4.50
165	部分生产力度量(编码)	-4.50
166	规模估算不足	-4.67
167	高级语言(过时)	-5.00
168	团队之间沟通不足	-5.33
169	变更控制不足	-5.42
170	价值分析不足	-5.50
最差实践		
171	团队成员之间的对立/摩擦	-6.00
172	成本估算方法不充分	-6.04
173	风险分析不足	-6.17
174	低级语言(过时)	-6.25
175	政府强制(时间紧迫)	-6.33

(续)

	方法、实践、结果	平均
176	测试不足	-6.38
177	管理者之间的对立 / 摩擦	-6.50
178	和项目干系人的沟通不足	-6.50
179	质量的度量不足	-6.50
180	问题报告不完整	-6.67
181	应用中易错的模块	-6.83
182	项目干系人之间的对立 / 摩擦	-6.83
183	需求变更评估失败	-6.85
184	缺陷跟踪方法不充分	-7.17
185	由于商业原因, 拒绝评估	-7.33
186	裁员或核心人员流失	-7.33
187	审查不完整	-7.42
188	安全控制不足	-7.48
189	进度的压力过大	-7.50
190	缺乏进度跟踪	-7.50
191	诉讼 (违反竞业禁止)	-7.50
192	成本跟踪不足	-7.75
193	诉讼 (违反合同)	-8.00
194	潜在缺陷 > 6.00/ 功能点	-9.00
195	可重用性 (高缺陷数目)	-9.17
196	缺陷去除效率 < 85%	-9.18
197	诉讼 (质量差 / 伤害)	-9.50
198	诉讼 (安全漏洞的危害)	-9.50
199	诉讼 (专利侵权)	-10.00
200	诉讼 (知识产权窃取)	-10.00

第7~9章将会讨论并且评估候选的最佳实践。第1章只概述本书的内容。

需要注意的是, 影响因素是多种多样的。它们包括完整的开发方法, 如 TSP (团队软件过程); 以及部分的开发方法, 例如 QFD (Quality Function Deployment, 质量功能展开); 包含具体的实践, 如各种类型的“审查”; 包括社交问题, 如项目干系人和开发者之间的不和谐; 包括称为有害因素的指标, 如“代码行”指标, 因为这个指标并不适合高级编程语言, 并且还扭曲了质量数据和生产率数据。当然, 所有这些因素都有一个共同的特点, 那就是它们要么提高质量和生产率, 要么降低质量和生产率。

因为编程语言也很重要, 所以你可能会问为什么本书没有涉及具体的编程语言, 如 Java、Ruby 以及 Objective C。那是因为, 截至 2009 年, 问世的编程语言已经超过了 700 种, 差不多平均每个月就会有一门新的语言诞生。

除此之外, 大多数的大型软件应用同时使用好几种编程语言, 例如, Java 和 HTML, 或者在相同的应用程序中, 组合使用当前最流行的十几种语言。第8章将会探讨语言对软件的影响以及语言的优缺点。但是由于语言的种类过于繁多, 并且它们的变化过于迅速, 因此要想对软件进行有效的评估, 我们至少需要几个月的时间。另外, 不管是高级语言还是低级语言, 并且不管是当前流行的语言还是已经“过时”的语言, 表 1-4 中只涵盖常规方式下的语言。

本书不是任何特定产品或方法的营销工具，只列举作者开发的一些工具和方法。另外，本书试图保持客观性，并且通过量化数据来获得结论，而不是通过主观意见。

为了说明方法和实践因项目规模而存在差异，表 1-5 展示了排名前 30 的最佳实践，左边的最佳实践适应于小型应用（拥有 1000 个功能点），右边的最佳实践适应于大型系统（拥有 10 000 个功能点）。正如我们所看到的，这两个表之间有很大的差异。

对于小型项目，敏捷开发、极限编程以及高级语言都是关键的实践，因为对小型应用程序来说，编码是主要活动。但是，当我们分析大型应用时，我们会发现质量控制才是重中之重。当然，详细的需求、设计以及架构对应用来说也相当重要。

应用程序的类型不同，它们的最佳实践也存在差异。表 1-6 分别罗列了信息技术（IT）项目和系统/嵌入式软件项目的前 30 个最佳实践。

虽然，对 IT 项目和系统/嵌入式项目来说，高质量可重用组件都是排名第一的最佳实践，但是两个表中剩余的最佳实践就截然不同了。对于信息技术项目，至少对于 500 强公司的开发来说，治理是排在第二位的最佳实践。这是因为管理的不称职可能会导致针对企业人员的刑事指控，如 2002 年的《萨班斯-奥克斯利法案》。

表 1-5 拥有 1000 和 10000 功能点软件项目的最佳实践

小型应用程序（1000 个功能点）		大型应用程序（10 000 个功能点）	
1	敏捷开发	1	可重用性（> 85% 零缺陷材料）
2	高级语言（当前）	2	潜在缺陷 < 3.00/ 功能点
3	极限编程（XP）	3	正规的成本跟踪报告
4	个体软件过程（PSP）	4	审查（需求）
5	可重用性（> 85% 零缺陷材料）	5	正规的安全分析
6	自动静态分析	6	缺陷去除效率度量
7	时间盒短迭代	7	团队软件过程（TSP）
8	可重用的源代码（零缺陷）	8	功能点分析（高速）
9	可重用的功能担保	9	能力成熟度模型集成（CMMI 5 级）
10	可重用的功能认证	10	自动化安全测试
11	潜在缺陷 < 3.00/ 功能点	11	审查（设计）
12	可重用的功能变更控制	12	缺陷去除效率 > 95%
13	可重用的功能召回方法	13	审查（代码）
14	面向对象（OO）开发	14	自动规模估算工具（功能点）
15	审查（代码）	15	混合型（CMM + TSP/PSP + 其他）
16	缺陷去除效率 > 95%	16	自动静态分析
17	混合型（CMM + TSP/PSP + 其他）	17	个体软件过程（PSP）
18	Scrum 会议（每日）	18	自动化的成本估算工具
19	缺陷去除效率的度量	19	需求变更的测量
20	功能点分析（IFPUG）	20	面向服务架构（SOA）
21	自动维护工作台	21	自动化质量与风险预测
22	提前评估潜在缺陷	22	行业数据的基准
23	团队软件过程（TSP）	23	质量功能展开（QFD）
24	用户参与团队工作	24	大型应用的正规架构

(续)

小型应用程序 (1000 个功能点)		大型应用程序 (10 000 个功能点)	
25	行业数据的基准	25	自动化缺陷跟踪工具
26	度量缺陷的严重级别	26	可重用架构 (可扩展)
27	用例	27	正规的风险管理
28	可重用的测试用例 (零缺陷)	28	项目活动级生产力度量
29	自动化安全测试	29	正规的进度报告 (每周一次)
30	不良修复的注入度量	30	功能点分析 (模式匹配)

表 1-6 IT 项目、嵌入式或系统项目的最佳实践

信息技术 (IT) 项目		系统 / 嵌入式项目	
1	可重用性 (> 85% 零缺陷材料)	1	可重用性 (> 85% 零缺陷材料)
2	正规的治理	2	潜在缺陷 < 3.00/ 功能点
3	团队软件过程 (TSP)	3	缺陷去除效率 > 95%
4	个体软件过程 (PSP)	4	团队软件过程 (TSP)
5	敏捷开发	5	度量缺陷的严重级别
6	缺陷去除效率 > 95%	6	审查 (代码)
7	正规的安全分析	7	精益六西格玛
8	正规的成本跟踪报告	8	软件六西格玛
9	潜在缺陷 < 3.00/ 功能点	9	自动静态分析
10	自动静态分析	10	缺陷去除效率度量
11	缺陷去除效率度量	11	混合型 (CMM+ TSP/ PSP+ 其他)
12	功能点分析 (IFPUG)	12	个体软件过程 (PSP)
13	面向服务的架构 (SOA)	13	正规的安全分析
14	联合应用设计 (JAD)	14	正规的成本跟踪报告
15	功能点分析 (高速)	15	功能点分析 (高速)
16	自动规模估算工具 (功能点)	16	审查 (设计)
17	通过数据挖掘来提取业务规则	17	自动化的项目管理工具
18	行业数据的基准	18	正规的测试计划
19	混合型 (CMM + TSP/PSP + 其他)	19	质量功能展开 (QFD)
20	可重用的功能认证	20	自动化成本估算工具
21	可重用的功能变更控制	21	自动安全测试
22	可重用的功能召回方法	22	面向对象 (OO) 开发
23	可重用功能担保	23	审查 (测试材料)
24	可重用的源代码 (零缺陷)	24	敏捷开发
25	提前评估潜在缺陷	25	自动规模估算工具 (功能点)
26	不良修复的注入度量	26	可重用的功能认证
27	可重用的测试用例 (零缺陷)	27	可重用的功能变更控制
28	检查 (要求)	28	可重用的功能召回方法
29	基于项目活动的生产力度量	29	可重用的功能担保
30	可重用的设计 (可扩展)	30	可重用的源代码 (零缺陷)

对于系统/嵌入式软件,多种形式的质量控制度量是排名最靠前的最佳实践。在软件史上,系统/嵌入式软件拥有最好、最先进的软件质量控制。这是因为系统和嵌入式领域的主要产品是复杂的物理设备,如果缺乏质量控制,将会导致灾难性的损害和毁灭。因此,医疗设备、飞机控制系统、燃油喷射系统以及其他形式的系统/嵌入式应用长期以来都有先进的质量控制,甚至在将软件使用在物理设备上之前,也必须进行质量控制。

最重要的一点就是,软件开发不是一个“一刀切”的工作。在开发中,我们必须仔细选择最佳实践,以匹配软件的规模和类型。

少数的基本原则能适用于各种规模和类型项目,质量控制、变更控制、好的评估以及好的度量都是至关重要的活动。可重用性也相当重要,需要说明的是,只有零缺陷的可重用对象才能提供可靠的价值。

虽然本书主要是关于软件工程的最佳实践的,但是本书也讨论了最佳实践对立的一面,即展示了最差实践。本书对最差实践的定义是,在大量的项目实践中证明一种方法或途径是有危害的。“危害”这个词的意思是,降低质量,降低生产率,或者隐瞒项目的真正状态。除此之外,“危害”还包括由于数据不准确而导致关于经济价值的结论错误。

大量的应用已经证明,个别的方法和路径是有害的。这并不是说,这些方法和途径总是无用的,只有在极少数情况下,这些方法才是有益的;但是多数情况下,它们对项目是有害的。

在软件行业里,有一个令人沮丧的消息,那就是糟糕的实践从来祸不单行。从软件项目的诉讼证词和法庭文件来看,这些项目往往是已取消的项目或者从来没有有效运行过的项目,并且这些项目还有一个共同点,那就是它们都同时使用了多个最差实践。

从软件方法和实践的使用模式来看,数据和观察的结果是最令人痛心不已的,那是因为,有害的实践或者最差实践竟然出现在美国65%的软件项目中。相反,在美国,只有14%的软件项目尝试使用9分以上的最佳实践。因此,对于大型应用来说,失败多于成功也就不足为奇了。

从笔者作为专家证人亲眼目睹的大量违反合同的诉讼中,笔者发现许多有害的实践往往会重复出现。在笔者看来,这些有害的实践都是造成“专业过失”的始作俑者,而专业过失最终会导致危害;但是一个受过训练的从业者应该知道什么是有害的,从而极力避免伤害。表1-7展示了大家都公认的30个有害实践。

这30个有害实践并不是同时出现。实际上,其中一些有害的实践,例如,对于拥有10 000个功能点的大型应用来说,使用通才只会有害。然而,收集的这些有害实践显然已经成了软件工程界的一个祸根,并且还引发了大量诉讼。

请注意,“代码行”和“平均缺陷成本”两个常用的指标称为“专业过失”。之所以将它们称为“专业过失”,是因为它们违背了标准经济学原理,并且也篡改了数据,使得人们不能看到经济结果。代码行指标显然不适合用于评估高级语言,而平均缺陷成本指标也不适合评估质量,但是平均缺陷成本指标却把错误最多的项目评为最好。这些问题将在后面进行更加详细的说明。

表 1-7 称为“专业过失”的软件方法和实践

排名	方法与实践	评分	排名	方法与实践	评分
1	缺陷去除效率 <85%	-9.18	16	风险分析不足	-6.17
2	潜在缺陷 > 6.00/ 功能点	-9.00	17	成本估算方法不完整	-6.04
3	可重用性 (高缺陷率)	-7.83	18	价值分析不充分	-5.50
4	成本跟踪不足	-7.75	19	变更控制不充分	-5.42
5	进度压力过大	-7.50	20	缺乏规模评估	-4.67
6	缺乏进度跟踪	-7.50	21	部分生产力度量 (编码)	-4.50
7	安全控制措施不足	-7.48	22	代码行 (LOC) 度量指标	-4.50
8	审查不充分	-7.42	23	治理不充分	-4.17
9	缺陷跟踪方法不足	-7.17	24	需求收集不充分	-3.67
10	未能评估需求变更	-6.85	25	平均缺陷成本度量指标	-2.83
11	应用中易错的模块	-6.83	26	客户支持不充分	-2.83
12	缺乏问题报告	-6.67	27	生产力度量不给力	-2.67
13	质量度量不充分	-6.50	28	大型系统的通才而非专家	-2.50
14	出于商业原因, 拒绝评估	-6.50	29	大型系统的人工成本估算方法	-2.50
15	测试不充分	-6.38	30	测试库控制不严格	-2.00

开发和维护软件有成千上万种可以使用的方法与技术, 但是并不是所有的方法和技术都能分为最佳实践或者最差实践。实际上, 对许多的实践和方法, 结果是如此模棱两可, 以至于它们经常称为“中等实践”。

中等实践的定义是, 一种工具或方法要么有助于软件开发要么阻碍软件开发。换句话说, 就是从使用该方法后, 无论是正方向还是负方向, 都没有量化的改变。

也许最有意思的观察是, 中等实践主要出现在小型项目 (不超过 1500 个功能点) 中。多年的数据表明, 小型项目能够以一种相对非正式的方式来开发, 并且最终的结果还是令人满意的。这种情况应该是大家习以为常的, 因为同样的情况适应于很多的产品。例如, 一个划艇并不需要和豪华游艇同样严格的开发过程。

因为小型项目远远超过了大型项目, 达到了 50 : 1 的比例, 因此, 如果小型项目是唯一适合分析的项目, 那么我们对最佳实践进行分析将会相当困难。有许多路径可以导致小型项目成功。但是随着应用程序规模的增加, 应用成功的路径却在减少。另外, 大学的研究很少有机会接触大型软件应用 (10 000 ~ 10 0000 个功能点) 的数据。因此, 当涉及软件工程的方法和结果时, 大学的研究与业界会大相径庭, 相差甚远。

认真分析最佳实践和最差实践需要从软件应用中获得数据, 并且这些应用的规模在 10 000 个功能点以上。对于大型应用, 失败远多于成功。在规模范围的两端, 最佳和最差实践的影响被放大。伴随着规模的增长, 在项目走向成功的路上, 质量控制、变更控制以及项目卓越的管理变得越来越重要了。

小型应用有许多优势。由于开发时间较短, 因此需求变更的数量也很少。通过采用多种的方法和过程, 我们可以成功地构建小型应用。例如, 对团队规模有限的小型项目来说,

评估和规划等方面会显得更加容易一些。

然而,创建大型软件应用(超过10 000个功能点)就要困难得多,并且项目也会有大量的需求变更。假如没有顶级的质量控制、变更控制以及项目管理,这个项目是不会成功的。大型应用需要多种类型的专家,并且有些专门的机构也是必需的,例如,项目办公室、正规的质量保证团队以及测试机构等。

1.8 总结

使大型软件项目夭折的路径有很多,但是使大型项目成功的路径寥寥无几。然而,对功能点少于1000的小型项目和功能点多于10 000的大型系统来说,导致成功的路径或“途径”是不相同的。在当今的社会中,对嵌入式应用、Web应用、商业软件应用以及许多其他类型的应用来说,从来没有两种应用成功的途径或者路径是一样的。

在软件开发实践中,最重要的是,在项目开始之前,就处理好规划和评估;在项目进行中,吸收不断变更的需求,并且成功地处理各种bug或缺陷。成功的另一个关键因素是,积极主动地发现问题并快速地解决问题,而不是忽视它们,并希望这些问题自动消失。

成功项目经常使用最先进的方法,并且在关键的活动上也是相当的优秀,例如,评估、变更控制、质量控制、进度跟踪以及问题处理等。相反,延期或者失败的项目往往都是因为有一个乐观的评估,没有预料到变更,没有严格的质量控制,没有及时的进度跟踪,并且忽略问题,以至于最终亡羊补牢,但悔之晚矣。

软件工程目前还不是一个真正被人们认可的工程领域,但是它不会永远是一个失败多于成功的领域。使软件工程进入一个真正的工程领域,将会有利于全球经济以及我们的专业地位。但是,要实现这些目标,我们就必须拥有更严格的质量控制、更妥善的变更控制、更优秀的度量方法,并且更加有效地量化我们的结果。

参考文献

这里介绍的阅读资料包括一些和软件最佳实践有关的书籍,也包括质量等更广泛的一些话题。其中一本参考文献是1982年普利策奖得主Paul Starr写的《The Social Transformation of American Medicine》(美国医学的社会转型)。这本书讨论了在医学教育和AMA(American Medical Association,美国医学协会)认证上的改进。遵循AMA的路径与改进软件工程教育、最佳实践以及认证有非常大的关联。Thomas Kuhn的著作《The Structure of Scientific Revolutions》(科学革命的结构)也包括在内。因为,如果把独特应用的个性开发转换成通过注册组件来构筑的通用应用,软件工程也会掀起一场革命。

Boehm, Barry. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981. (中文版《软件工程经济学》,李师贤等译,机械工业出版社2004年7月出版)

Brooks, Fred. *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1974, rev. 1995. (中文版《人月神话》,UMLChian 翻译组汪颖译,清华大学出版社出版)

- Bundschuh, Manfred, and Carol Dekkers. *The IT Measurement Compendium*. Berlin: Springer-Verlag, 2008.
- Charette, Bob. *Software Engineering Risk Analysis and Management*. New York: McGraw-Hill, 1989.
- Crosby, Philip B. *Quality Is Free*. New York: New American Library, Mentor Books, 1979. (最新中文版《质量免费》, 杨钢、林海译, 山西出版集团山西教育出版社 2011 年 6 月出版)
- DeMarco, Tom. *Controlling Software Projects*. New York: Yourdon Press, 1982.
- DeMarco, Tom. *Peopleware: Productive Projects and Teams*. New York: Dorset House, 1999.
- Garmus, David, and David Herron. *Function Point Analysis—Measurement Practices for Successful Software Projects*. Boston: Addison Wesley Longman, 2001. (中文版《功能点分析——成功软件项目的测量实践》, 钱岭、苏薇、盛铁阳译, 清华大学出版社 2003 年 12 月出版)
- Gilb, Tom, and Dorothy Graham. *Software Inspections*. Reading, MA: Addison Wesley, 1993.
- Glass, Robert L. *Software Runaways: Lessons Learned from Massive Software Project Failures*. Englewood Cliffs, NJ: Prentice Hall, 1998.
- Glass, Robert L. *Software Creativity*. Second Edition. Atlanta, GA: developer.*books, 2006.
- Hamer-Hodges, Ken. *Authorization Oriented Architecture—Open Application Networking and Security in the 21st Century*. Philadelphia: Auerbach Publications, to be published in December 2009.
- Humphrey, Watts. *Managing the Software Process*. Reading, MA: Addison Wesley, 1989. (中文版《软件过程管理》, 高书敬、顾铁成、胡寅译, 清华大学出版社 2003 年 3 月出版)
- Humphrey, Watts. *PSP: A Self-Improvement Process for Software Engineers*. Upper Saddle River, NJ: Addison Wesley, 2005.
- Humphrey, Watts. *TSP—Leading a Development Team*. Boston: Addison Wesley, 2006. (中文版《TSP-领导开发团队》, 人民邮电出版社, 2007 年 1 月出版)
- Humphrey, Watts. *Winning with Software: An Executive Strategy*. Boston: Addison Wesley, 2002.
- Jones, Capers. *Applied Software Measurement*, Third Edition. New York: McGraw-Hill, 2008.
- Jones, Capers. *Estimating Software Costs*. New York: McGraw-Hill, 2007. (中文版《软件项目估计》(第 2 版), 电子工业出版社, 2008 年 3 月出版)
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Boston: Addison Wesley Longman, 2000. (中文版《软件评估、基准测试与最佳实践》, 机械工业出版社, 2003 年 4 月出版)
- Kan, Stephen H. *Metrics and Models in Software Quality Engineering*. Second Edition. Boston: Addison Wesley Longman, 2003.
- Kuhn, Thomas. *The Structure of Scientific Revolutions*. Chicago: University of Chicago Press, 1996.
- Love, Tom. *Object Lessons*. New York: SIGS Books, 1993.
- McConnell, Steve. *Code Complete*. Redmond, WA: Microsoft Press, 1993.
- Myers, Glenford. *The Art of Software Testing*. New York: John Wiley & Sons, 1979. (最新中文版《软件测试的艺术》(原书第 3 版), 张晓明、黄琳译, 机械工业出版社 2012 年 4 月出版)
- Pressman, Roger. *Software Engineering—A Practitioner's Approach*, Sixth Edition. New York: McGraw-Hill, 2005. (中文版《软件工程: 实践者的研究方法》(第 6 版), 机械工业出版社, 2007 年 1 月出版。最新

中文版为原书第7版中译版,机械工业出版社2011年5月出版)

Starr, Paul. *The Social Transformation of American Medicine*. New York: Basic Books, 1982.

Strassmann, Paul. *The Squandered Computer*. Stamford, CT: Information Economics Press, 1997.

Weinberg, Gerald M. *Becoming a Technical Leader*. New York: Dorset House, 1986.

Weinberg, Gerald M. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971. (最新中文版《程序开发心理学》(银年纪念版),电子工业出版社2010年3月出版)

Yourdon, Ed. *Death March—The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*. Upper Saddle River, NJ: Prentice Hall PTR, 1997. (中文版《死亡之旅》,周浩宇、杨华译,机械工业出版社2012年1月出版)

Yourdon, Ed. *Outsource: Competing in the Global Productivity Race*. Upper Saddle River, NJ: Prentice Hall PTR, 2005.

50 个软件最佳实践概述

由于并不是每个人都从头到尾、仔仔细细地看书，因此在开始讨论本书的主题之前，我们似乎有必要对软件工程最佳实践进行简要概述。碰巧的是，本章是由一个官司引出的，不久这个官司就解决了。本书中有关最佳实践的素材已经更新，其中包括软件工程技术最近的一些变化。

这些最佳实践的讨论主要集中在规模为 10 000 个功能点的大型项目。这样做是非常有意义的，因为在这个范围内，延期和取消的项目逐渐超过了成功的项目。

本书中所讨论的最佳实践可以使用 30 年以上。开发大型应用大约需要五年的实践，部署和定制这样的应用可能又需要一年时间，而部署完毕之后，大型应用将具有很长的使用年限，一般会超过 25 年以上。

在 25 年的使用年限内，软件应用会有许多功能需要改进，并且还有许多缺陷需要修复。偶尔也可能会有应用的“改造”或重组，也许会更改文件格式，或者将应用的源代码转换成另一种新式语言等。

这里所讨论的最佳实践纵贯软件工程的整个生命周期，从项目开始直到应用停止服务。这里罗列了 50 个方面的最佳实践，但并未详尽：

1. 最大限度地减少裁员所带来的危害。
2. 技术人员的积极性和动力。
3. 经理和高管的积极性和动力。
4. 软件人才的选拔和招聘。
5. 软件人才的考核以及职业生涯规划。
6. 软件应用早期的范围控制。
7. 软件应用的外包。
8. 承包商和管理顾问的使用。
9. 选择软件的方法、工具以及实践。
10. 认证方法、工具以及实践。
11. 软件应用的需求。
12. 用户参与软件项目。
13. 软件应用的行政管理支持。
14. 软件架构和设计。

15. 软件项目的规划。
16. 软件项目的成本估算。
17. 软件项目的风险分析。
18. 软件项目的价值分析。
19. 取消或拯救陷入困境的项目。
20. 软件项目的组织结构。
21. 培训软件项目经理。
22. 培训软件技术人员。
23. 使用软件专家。
24. 软件工程师、专家以及管理人员的认证。
25. 软件项目中的沟通。
26. 软件的可重用性。
27. 可重用材料的认证。
28. 编程。
29. 软件项目管理。
30. 软件项目的度量和指标。
31. 软件的基准和基线。
32. 软件项目的里程碑和成本跟踪。
33. 软件发布前的变更控制。
34. 配置控制。
35. 软件的质量保证。
36. 审查以及静态分析。
37. 测试和测试库的控制。
38. 软件的安全性分析和控制。
39. 软件的性能分析。
40. 软件的国际标准。
41. 软件中的知识产权保护。
42. 病毒、间谍软件以及黑客的预防。
43. 软件的部署和定制。
44. 培训软件应用的客户或用户。
45. 软件应用部署后的客户支持。
46. 软件担保和召回。
47. 软件发布后的变更管理。
48. 软件的维护和功能增强。
49. 软件应用的更新和发布。
50. 遗留应用的终止。

以上是 50 个最佳实践的概述，主要涉及管理和技术领域。接下来，我们将对它们进行

深入探讨。

2.1 最大限度地减少裁员所带来的危害

在写本书的时候，全球经济正迅速陷入自大萧条以来最严重的经济危机中。因此，有很多员工被解雇。更糟糕的是，一些科技公司投资资金也即将告罄，而等待它们的将是破产。

通过对以往的经济衰退进行研究，笔者发现企业在处理裁员时，往往会犯严重的错误。首先，由于要解雇的人员通常是由经理和高管确定的，因此下岗的技术人员往往会比管理人员多，这样做显然会降低公司的运营绩效。

其次，行政和支持人员会比软件工程师和技术人员提前被裁掉，这些行政和支持人员包括：质量保证（Quality Assurance）、技术作家（Technical Writer）、指标和度量专家、秘书、程序管理员等。在行政人员被解雇之后，技术人员不得不承担各种各样的行政工作，然而他们并没有受过特定的训练，因此降低公司的运营绩效也是必然的了。

严重的裁员带来的后果是，在随后的几年里，公司的生产率和质量会有所下降。尽管目前还没有较为完美的方法来处理大规模的裁员，但是以下的一些方法或许可以最大限度地减少裁员所带来的损失。

引进转职推荐服务，以帮助员工创建简历，并且如果有合适工作的话，尽量帮助他们。

对于有多个办公地点的大公司，尽量在本公司提供就业机会。笔者曾经碰到过一个大公司，该公司的两个部门位于同一栋楼，其中一个部门裁掉的员工，竟然被另一个部门给雇用了，然而两个部门事先并没有进行过任何形式的沟通。

如果你的公司是一家美国公司，雇用的员工是获得美国临时签证的海外员工，那么在经济衰退期间，解雇本地（美国）员工的数量远高于海外员工的数量将是很不明智的。更糟糕的是，裁减本地（美国）员工，以招聘海外员工。像微软和英特尔这样的大公司已经有了前车之鉴，这样做会导致员工的士气严重下降，并且也会对公司的形象产生负面的影响，除此之外，还可能招致州政府和联邦政府的调查。

优先考虑正在开发的应用以及积压的应用，并尝试削减投资回报率较低的应用。

分析遗留应用的维护状况，并考虑在不降低安全性或运营业绩的情况下，减少质量改进方面的维护人员的数量，如应用的改造和重组、易错模块的去除等。

在削减投资回报率较低的应用之后，需要对员工的编制模式进行计算，并处理积压的应用以及正在开发的应用。

在削减时，需要考虑提高经理的控制范围，或者增加经理所管辖的技术人员的数量，如将经理的控制范围从 8 名员工提高到 12 名员工。事实上，要想减少大型项目经理之间的争用和纠纷，较大的控制范围可能会奏效。

不要试图在审查、静态分析、测试以及质量控制等活动上吝啬。高品质会产生更好的性能以及更精悍的团队，而低质量则会导致工期延误、成本超支以及其他问题，并且最终的回报也很小。

仔细分析专家在技术人员中所占的比例，专家的类型包括技术写作、质量保证以及配

置管理等。解雇大量的专家，将会降低软件工程师的运营性能。

在严重的经济衰退中，一些即将离职的员工可能是团队核心成员，他们对大量的产品、发明创造以及知识产权了如指掌。虽然大多数公司通过签订保密协议来保护公司的合作权益，但是很少有公司尝试创建一个知识库，用以存储可能会离职的核心人员。如果公司采用的是礼貌而又专业的方式来处理裁员，那么大多数的员工将很乐意留下关键的信息。这可以通过问卷调查或“知识”采访的形式进行。但是，如果裁员的时候，公司对员工很粗暴也很无情，那么我们就不要指望员工在离职后，会留下很多关键的信息。

只有在极少数情况下，公司的研究设施才是完全封闭的，有些公司允许离职员工获得知识产权的权利，如版权和员工申请的专利。我的想法是，一些员工在离职后可能会创建新公司，从而在有用的点子上继续取得进展，不然的话他们将会淡出我们的视线。

当开始削减员工的时候，我们需要考虑软件组织的典型工作模式。对于一个有 1000 名员工的公司，技术人员（如软件工程类的技术人员）占了一半左右，各类专家占了 30% 左右，而剩余的 20% 则是管理和辅助人员。然而，员工们大多数的时间和精力都花在寻找和修正错误上，而其他可度量的活动则投入很少。

裁员后，采用提高产品质量的技术可能会更加有利，更少的员工可能会做出更富有成效的工作。因此某些工作可能允许减少人手以获得更高的生产率，这些工作包括：需求设计、代码审查、六西格玛、静态分析、自动化测试以及一些强调质量控制的方法等，如团队软件过程（TSP）。

作者在 2005 年对工作模式进行了研究，在一年 220 天的正常工作中，软件技术人员用来开发新应用的时间只有 47 天，用来做测试和缺陷修复的时间约有 70 天，而剩余的时间则是用在会议、管理任务以及不断变更的需求上。

因此可以将缺陷预防与更有效的缺陷去除（即审查和测试、自动化测试、静态分析等）相结合，这在减少工作量的同时还提高了产品的质量。如果有可能，可以将缺陷去除的天数由 70 天减少为 20 天，这样的话，我们就可以在新功能的开发上再投入一倍的时间了。

通常在经济衰退期间，客户支持是最先遭到裁减的，因此可想而知，产品会在客户满意度上出现严重的问题。在交付之前，高质量的产品将允许较小规模的客户支持团队来处理更多的客户。由于客户支持主要集中在缺陷上，因此可以推测，在已交付应用的 220 个缺陷中，每减少一个缺陷就可以减少一个客户支持人员，但这样却并不会降低响应时间，也不会降低客户的满意度。以上假设的前提是，客户支持人员每天大约和 30 个客户说话，并且每一个发布的缺陷会被 30 个客户遇到。因此，每一个发布的缺陷会占用一个客户支持人员一天的时间，并且每年有 220 个工作日。

另一种可能的解决办法是改造遗留应用，而非构建新的替代品。通过对遗留应用执行一系列的操作：改造和消除容易错模块；执行静态分析工具；重组高度复杂的代码；将代码转换成另一种新语言。这样可能会将遗留应用的使用年限延长五年以上，同时在部署之前，每移除 120 个错误，可以减少一个维护人员。这个假设的前提是，负责维护的程序员每个月一般可以解决 10 个错误（严重的错误为 1 到 2 个）。

最根本的是，如果美国的质量控制比今天更好，那么较少的员工就可以完成更多新应

用的开发。在测试之前，有许多错误既可以被预防也可以被移除，然而大量的时间都花在缺陷去除上面了。

通过审核、静态分析、自动化测试以及传统测试等手段，将缺陷预防与有效的缺陷去除相结合，可以减少 25% 的开发人员、50% 的维护人员以及 20% 的客户支持人员，然而在运营效率、顾客的满意度以及工作效率上却没有丝毫减少。事实上，应该改进开发计划，因为缺陷太多，导致人们通常在测试上花费了过多的时间。随着经济衰退的深化，我们不仅有必要记住 Phil Crosby 所说的：“质量是免费的”，而且也应该明白这也为软件提供了显著的经济效益。

在软件行业有一个问题已经存在了多年，那就是，我们几乎没有任何可靠的经济研究，并公布有力的证据来证明软件质量是有价值的。之所以如此，是因为代码行和平均缺陷成本两种常见的度量指标漏洞百出，已经不再适用于处理经济议题了。在每个功能点上使用缺陷去除成本法是一个很好的选择，但是这些度量指标需要部署到组织中去，并且与此同时，还得不断地积累工作量、成本以及质量数据等。从作者所进行的研究来看，将缺陷预防和缺陷去除的方法相结合，可以降低潜在缺陷，使得缺陷去除效率达到 95% 以上，同时也可以降低开发成本、开发周期、维护成本以及客户支持成本等。

除了以上提到的裁员，许多企业都开始减少正规工作月的数量，并且还要求部分员工实行无薪休假，以保持现金流正常运行。减少工作时间和降低所有员工的福利待遇可能比削减人员所带来的危害小，并且还能确保其他的员工有持续的福利待遇。然而，谨慎实行也是很有必要的，因为如果要求员工休假的天数超过了一定限度，那么员工可能会认为自己在法律上已经由一个专职员工变成了一个兼职员工。如果这种情况发生，那么他们的医疗福利、养老金以及其他的一些企业特殊待遇则可能被终止。虽然公司与公司之间的政策各不相同，没有一个放之四海而皆准的政策，但是这确实是一个不容忽视的问题。

许多州政府和市政府的信息技术员工早就不再享受由公司提供的福利了。这些福利包括补偿未使用的病假、既定的退休计划、积累的年假以及免费的医疗福利等，并且在员工退休的时候，政府还会将员工未休的假期折成钱支付给他们。由于州政府正在面临着赤字，因此这些特殊利益在未来很可能会消失。

目前在裁员上依然没有任何完美的解决方案，但是削减大量的专家和行政人员很可能会导致意想不到的问题。此外，更好的质量控制以及更好的维护或改造可以减少员工的实际数量，同时无需过度加班，并且也不会降低运营效率和客户满意度。

2.2 技术人员的积极性和动力

许多软件工程师和专家（如质量保证和技术作家）都是精力充沛，并且很有上进心的人。软件人才心理学的研究说明了一些有趣的现象，如高离婚率和内向倾向。

拖延时日似乎已经成了软件开发和维护的天性了，并且有时甚至会让员工加班熬夜。话虽这么说，但是，还是有许多方法可以让技术人员动力十足、士气高昂的。

通过对大公司软件工程师的离职面谈进行研究，我们发现了两个令人极为苦恼的问题：

(1) 最优秀的人才往往不愿意待在人多的地方；(2) 自愿离职最常见的原因是：“我不喜欢糟糕的管理。”

因此，一些先进企业（如 IBM）已经开始尝试反向考核了，即员工对领导的管理绩效以及正常的考核进行评估，而领导对员工的绩效进行评估。

请注意，以下的主题是一些士气高昂的龙头企业提出的，如 IBM、微软、谷歌等，这些主题包括：

强调做正确的事情，而不是仅仅通过延长工作时间，以赶上项目的进度。

如果个人觉得某项目是有价值的，公司允许支持一些个人项目。

确保淘汰掉边缘的管理人员，因为管理不善会导致大量优秀的软件工程师离职。

确保考核公平、公正，如果员工认为他们由于某种原因被无端地降级了，他们可以提出申诉。

让管理人员和技术人员偶尔有一个早餐或午餐会议，以便他们能够在开放宽松的环境下就共同关注的议题展开讨论。

有一个正规的申诉或“开后门”的渠道，让技术人员觉得他们在得到了不公平的对待后，可以向更高级别的管理者申诉。实施这个计划的一个重要前提是“无报复”。也就是说，不能惩罚提出申诉的员工。

偶尔奖励一下工作出色的员工，并且请注意，许多小的奖项可能比一个大奖更有效，如“双人晚餐”或休息日等。但是，坚决不奖励那些以牺牲质量为代价来获得生产效率或进度的员工。

当公司的业务或经济状况发生改变时，确保让所有的技术人员知道公司到底发生了什么。如果一家公司的财务陷入困境或即将被并购，那么通过正式会议来保持人员的斗志是难能可贵的。

对提议的方案进行实际的评估，并采取行动，这往往是有益的；但是，倘若没有对提议的方案采取任何行动，那么这往往是有害的。

令人不解的是，加班往往能提高员工的士气。加班能使项目看起来更有价值，否则员工们不会要求加班。但连续加班几个星期（即 60 小时/周）则是有害的。

一个复杂的问题是，大多数公司的软件工程师被视为“专业工作人员”，而非钟点工。除非软件工程师和技术员工都是工会成员，不然他们无论工作多少小时，通常都不会获得任何加班费的。当然，从法律的层面上来讲，这个问题超出了本书的范围，因此在此不做讨论。

公司的培训和教育可以让员工拥有更高的士气以及更好的精神面貌。因此无论是内部培训或外部培训，每年至少预留 10 天的培训时间将是非常有益的。有趣的是，每年培训 10 天以上的公司比没有任何培训的公司有较高的生产率。

这里给出了一些基本的思路，除了以上这些因素以外，也有其他一些因素会影响员工的士气。公平、沟通并有机会做创新性的工作都是提高软件工程师士气的关键因素。

由于全球经济已经陷入了严重的衰退中，因此就业机会将变得很稀缺，甚至只留给那些表现最佳的员工。毫无疑问，由于公司在走下坡路，所以员工的福利也将会随之消失。

经济衰退和经济危机可能会引入新的未知因素。

2.3 经理和高管的积极性与动力

经济史学家将 1908 年到 2008 年之间近百年的时间称之为高管们薪酬和福利的“黄金时代”。

伴随着全球的金融危机和经济衰退，各个公司为了摆脱行业困境，都使出浑身解数，无所不用其极，于是他们都纷纷抛出了一个令人不安的话题焦点，即高管们非凡的薪金、奖金以及退休福利。

许多行业的高管不仅有数百万美元的年薪，还有数百万美元的奖金、股票、退休金、终身福利以及“金色降落伞^①”福利待遇协议。

其他福利还包括使用公司的专用飞机、豪华轿车以及大型体育场馆的豪华包厢等，另外还可以成为健康俱乐部的会员以及高尔夫球会的会员等。

从理论上讲，高管获得这些福利是应该的，因为高管都为实现股东的最大利益而不断努力，另外还不断拓展商机，引导企业不断走向成功。

但由于全球的金融危机以及经济衰退，再加上大量高管的欺诈和舞弊，公司可能会杜绝高管无限制的补偿和福利。在全球经济衰退的大潮中，成千上万的公司都在赔钱，甚至正在走向破产。在美国，至少接受联邦政府“保释”金的公司将对高管的薪酬进行限制，而其他一些公司也正在酝酿着对高管的薪酬进行限制。

从 2008 年起，高管们的薪酬将逐渐成为下一个公众关注的焦点，并且他们的薪酬很可能会取决于企业的营利能力以及业务的成功度。我们希望，在金融危机过后，企业能更仔细地构想决策，并且高瞻远瞩，从长计议，谨慎考虑长远的后果。

在各级行政总裁和高级副总裁下面是成千上万的一线、二线、三线管理人员、董事会成员以及公司的其他管理成员。

在这些低层次的管理中，软件工程师和技术人员的待遇类似于高管们的薪酬。事实上，在一些企业中，首席软件工程师的薪酬比一线、二线的经理还多，当然，这是因为他们付出的比较多。

在软件应用中，成功的管理往往是将管理能力与技术能力相结合。许多软件经理都是从软件工程师转到管理的，那是因为他们有突出的解决问题的能力，以及出色的领导才能。

有一个微妙的问题值得讨论一下。如果每个经理的控制跨度或向经理报告的技术人员的数量接近全国的平均水平 8，那么无论如何合适的工作，我们都很难找到合格的管理者。换句话说，合适的控制范围是让 12.5% 的员工处于领导岗位，但小于 8% 将恰到好处。

提高管理者的控制范围，并且让不合格的管理者走下管理岗位，将是明智之举。实行这项政策的关键问题是，管理者怎么才知道这么多员工的表现呢？然而，在目前的控制范围下，管理人员往往会花费更多的时间与其他经理开会，而非和自己管理的人在一起。

① 金色降落伞 (Golden Parachute, 又译黄金降落伞)，是按照聘用合同中公司控制权变动条款对高层管理人员进行补偿的规定，最早产生于美国。

截至2009年,软件项目管理依然是最棘手的管理工作之一。软件项目经理往往担负着各种骂名,他们经常开会,并且将自己良好的愿望强加给员工,而他们自己往往则激情满怀,乐此不疲。

当软件项目失败或延期的时候,即使有些问题是由于更高级别的进度约束或客户无理的要求所造成的,管理人员依然会受到大量责备。因此这对软件项目经理来说显然是一个很不幸的事实:他们为项目的失败承担了大部分的责任,并且与硬件工程经理、营销经理以及其他管理人员相比,他们分享的荣耀却很少。

软件项目管理所面临的主要问题有:进度约束、成本约束、需求蔓延、质量控制、进度跟踪以及人事问题等。

要想精确估算软件项目的精度是相当困难的。实际上,大多数的软件项目都会出现延期,而延期的时间长短往往与应用程序的规模有关。因此在持续的进度压力之下,员工的士气会受到一定的影响。解决这种问题的一个办法是,通过查看历史数据以获得类似项目的日程安排。如果公司内部的历史数据不可用,那么可以从外部获得数据,如澳大利亚的国际软件基准组织(ISBSG)。细致的分工也是相当有用的。问题是,项目进度要与员工当前的士气匹配,二者时刻保持同步。除此之外,由于成本和进度是紧密联系的,因此进度同样也得和实际成本匹配。

大多数软件项目的成本和进度一般都会超过原先的估计和预算,其中一个主要的原因就是需求蔓延。通过使用功能点来对需求和规范进行度量,我们发现每个月的平均蔓延率约为2%。一旦知道了这个规律,就可以在初始估算的时候加入需求蔓延。在任何情况下,一旦需求发生显著变化,都会重新估算项目的进度和成本。如果不这样做,将会导致严重的超支,甚至经营信誉下降,当然信誉度受损也会影响员工的士气。

大多数的软件项目在测试过程中,由于过多的缺陷,不得不将进度推迟。因此,前期的缺陷预防和缺陷去除活动(如审查和静态分析)将是解决进度延期和成本超支最为行之有效的方法。不幸的是,并不是许多经理都认可这一点,大多数的经理往往吝质量控制。但是,如果质量控制得好,员工也将激情满怀、斗志昂扬,同时该项目也将顺利地达到预期目标。因此,卓越的质量控制往往会有利于保持经理以及技术人员的士气。

跟踪软件项目进度以及报告问题也许是软件项目管理中最薄弱的环节。在许多违反合同的诉讼中,我们通过证言发现,技术人员和一线的管理人员对已经存在的严重问题心知肚明,但他们却并没有将这些问题及时告知更高级别管理人员或客户,而是当不得不解决这些问题的时候,他们才发现亡羊补牢,悔之晚矣。项目跟踪的基本准则应该是:“没有任何惊喜。”倘若碰到问题,不去解决,问题是不会自动消失的,所以一旦发现问题,就该立即上报,并竭尽全力解决它们。这样做将有利于保持员工和管理人员激昂的士气。

人事问题也是软件项目中相当重要的问题。由于许多软件工程师的上进心较强,具有较高的斗志,并且也具有一定的创新能力,因此标杆管理(Management By Example)比法令管理(Management By Decree)强很多。管理者需要给员工公平、公正的考核,不偏私、不袒护,并确保这些员工能够知道所有来自公司高层的通知,如裁员或业务单位的销售额等。

不幸的是,软件项目中员工的士气与软件项目的成功息息相关,截至2009年,有许多

项目都以失败告终。规划和预算以历史数据和基准为基础，而非客户的需求，这也将有助于提高员工的士气。与估算相比，历史数据更值得信赖。

2.4 软件人才的选拔和招聘

由于全球的经济都陷入了严重的危机，因此许多公司都在裁员，有的公司甚至都歇业了。然而，这却给那些正在扩张的公司提供了一个绝佳的机会，他们可以乘机引进大量的人才。到 2009 年为止，就业市场上还没有出现过像现在这么多的软件人才。

当然，公司对应聘者进行背景调查也是相当重要的，虚假的简历并不少见，更何况当前正处于经济衰退期间，虚假简历可能会有所增加。此外，对管理和技术人员实施多方采访，将有利于应聘者更快地融入团队，并妥善应对即将开始的项目。

如果刚入职的员工正在考虑他们走出校门的第一份工作，那么某种形式的能力测试对他们来说也许很有用。有些公司还让工业心理学家对刚入职的员工进行心理访谈。然而，这些方法的效果并不是很明显。

多次访谈，并结合 6 个月一次的评估看起来似乎有很好的效果。评估期间的优异表现是他们加入团队的一个前提。

2.5 软件人员的考核以及职业生涯规划

工作五年后，软件工程师往往会面临自己职业生涯中的一项重大决定：要么继续做技术工作，要么进入管理层。

技术职业生涯之路可以是智力上的满足，并且在许多领先的公司也都有很好的福利待遇。如“高级软件工程师”、“企业研究员”以及“咨询架构师”等职位都很常见，并且这些职位也是大家心驰神往的。像 IBM 这样的大公司都有自己的研究部门，并且首席工程师可以做自己认为很有创意的软件项目。

虽然有些管理者继续进行技术工作，但是他们承担了更多的责任，如进度管理、成本管理、质量管理以及人事管理等，因此他们在技术上投入的时间量明显减少了很多。

软件工程有几个不同的职业生涯道路，如编程开发、编程维护、业务分析、系统分析、质量保证、架构以及测试等，它们都朝着不同的方向发展。

这些不同的专业方向说明了一个事实，那就是软件工程还不是一个真正的职业，而是一个没有被国家认可的职业。目前，有许多种自愿性的专业认证，如测试和质量保证，但它们都没有任何法律地位。

大型企业的软件组织可能会有 90 多种不同类型的专家，包括技术作家、软件质量保证专家、指标专家、集成专家、配置控制专家、数据库管理员以及程序库管理员等。然而，在不同公司里，这些专家的职业各不相同，并且他们也没有经过标准的训练，甚至也没有标准的定义。

许多公司都没有标准的职称，并且还使用一个通用的称呼，如“技术人员”，这样的技

术人员可能包含十几种类型的职业。

在对大公司的软件职业进行研究时,我们发现人力资源往往不知道该招什么方面的人才。因此,为了获得这些基本信息,人力资源的员工有必要到网上查询相关信息,或者向管理和技术方面的员工。

以往,对一个最优秀的技术人员和管理人员来说,一个好的职业生涯规划应包括“跳槽”,从一个公司跳到另一个公司。许多公司的内部政策都限制加薪,然而跳到另一家公司,却可以绕过这些限制。不过,当前的经济不景气,该方法也有些行不通了。当前,许多公司都冻结招聘,并试图减少员工数量,而不是扩充。事实上,有些公司甚至可能会走向破产。

2.6 软件应用早期的范围控制

多年来,预测软件应用的规模是相当困难的,并且估算结果也不是很准确。只有需求出来了,我们才能使用功能点指标来度量应用的规模,但对初始的软件成本估算和进度规划来说,那时已经太晚了。如果确实存在相似的应用,那么源代码的规模只能通过相似应用来获得。

然而,在2008~2009年之间,新型分析软件应用规模的方法已经出现了。如今,国际软件基准组织(ISBSG)已经到达了一个临界规模,拥有5000多个软件应用的历史数据,因此我们可以从ISBSG获得相似软件应用规模的可靠数据。

由于许多应用程序都与现有的应用程序非常相似,因此从ISBSG获得相似应用的规模数据已经成为项目早期的一个标准活动。要获得的数据还包括进度和成本方面的信息,它们甚至比应用程序的规模更有价值。然而,ISBSG的数据支持功能点指标而不是代码行指标。由于使用功能点指标是一个最佳实践,而使用代码行指标则是不恰当的,这当然不是一个坏的情况,但是对于那些坚持使用代码行指标的公司来说,它们会失去使用ISBSG度量基准的机会。

对于新型的软件或没有使用ISBSG数据表示的应用程序来说,目前几种快速估算应用规模的方法也许很合适。一种是基于模式匹配的新方法说明书,该方法可以获得功能点、源代码的近似规模,甚至还可以获得其他方面的信息,如规格的页数等。在开发过程中,这种方法还能预测需求的增长速度,然而预测需求增长速度一直是软件项目的薄弱环节。

其他估算规模的方法包括各种新型功能点近似或“轻量级”的功能点分析,这些方法可以在短短几分钟内预测出功能点的规模,而不是以正常的速度(每天大约400个功能点)来预测。

前期及时估算应用规模是准确估算的前提,同时也是风险分析的前提。许多风险都是与应用程序的规模成正比的,所以越早知道应用的规模,越能获得比较完整的风险分析。

对于拥有1000个功能点的小型应用来说,每个功能都会有一个对应的版本。然而,对于拥有1万到10万个功能点的大型应用来说,拥有多个版本就很正常了。

(小型项目一般使用敏捷开发,短期内能开发完成的单一特性或功能称为Sprint。这些功能或特性通常有100~200个功能点)。

由于项目的进度和成本与应用的规模成正比，因此大型系统通常会把系统划分成多个版本，差不多每 12 ~ 18 个月迭代一次。知道应用整体的规模以及单个功能和特性的规模，我们就可以指定一个有效的版本策略，该策略可能涉及三到四个连续的版本。了解每个版本的规模之后，准确地估算项目的进度和成本就显得很容易了。

在获得需求之前，我们可以通过模式匹配来获得应用的规模，这种方法是先获得软件应用的外部描述，然后再根据描述来匹配其他相似应用。

快速功能点法在时间上会有些出入，要想准确地估算应用的规模，至少需要获得应用的部分需求。

在软件应用遇到资金问题之前，估算应用规模的最佳实践是使用一个或多个（或全部）快速估算应用规模的方法。如果应用的规模足够大，那么项目可能会存在很大的风险，因此当时间充裕时，可以在开发前采取适当的纠正措施。

两种新型软件范围控制方法最近开始走进人们的视野，它们貌似都很有效。一种起源于芬兰的方法被称为北方范围估算法。另一种起源于澳大利亚的方法被称为南方范围估算法。两种方法大致是相似的，它们都试图尽早估算应用的规模，并任命一个正式的范围经理，以检测可能出现的新功能。通过不断地关注范围和增长的问题，使用这些方法的软件项目在它们最初的版本中有更高的成功率，而不是将许多功能都积压在后续的几个版本中。

这些范围控制的新方法实际上产生了一个新的职业——范围经理。这个新的职业与其他几个新职业在过去的几年里已经出现，如网络管理员和 Scrum Master。

近年来，规模估算能力已经有所提高，并且 ISBSG 的基准与新型快速规模估算法相结合将会在范围控制上做出更大的改善。

2.7 软件应用的外包

在过去的 20 年里，美国的公司已经接触到了一个主要的业务问题：软件应用应该由内部开发，还是应该交给承包商或外包开发。事实上，这个问题所涉及的范围比单个的应用广得多，它可以涵盖所有的软件开发业务、所有的软件维护业务、所有的客户支持业务或者是整个软件组织。

在外包协议中，最佳实践的必要性通过以下这个事实得到了很好的体现，即，在短短的两年时间里，大约有 25% 的客户和外包供应商产生了矛盾。尽管客户不同，得到的结果也不尽相同，但是在美国，外包总体预测结果的近似分布如表 2-1 所示，该表数据来自于笔者对众多客户的观察。

软件开发和维护的代价昂贵，并且已经成为了企业预算的主要组成部分。软件专业人士的数量超过公司员工总数的 5% 并不少见，并且软件和计算机的总预算超过企业年度支出 10% 的情况也很常见。

表 2-1 24 个月，美国软件外包情况的近似分布

结果	外包协议所占的比例
双方合作愉快	70%
客户或供应商有一些不满	15%
按计划解除协议	10%
客户和承包商之间诉讼的可能性	4%
客户和承包商之间正在打官司	1%

假如使用功能点指标作为度量的基础,那么在大型机项目组合中,大公司所有软件的功能点总数可能多达250万,而一些非常大的公司,如AT & T和IBM,它们拥有的功能点总数可能已经超过1000万了。

在现代商业中,软件和软件人才的增长都是不可预知的。例如,现在一些大规模的银行和保险公司有数以千计的软件人员。事实上,软件和计算机技术人员已经成了许多公司最大的单一职业群体,但他们的核心业务却不是软件。

由于软件的使用越来越广泛,并且软件的成本也越来越昂贵,因此许多大型企业的高管都在问一个基本的问题,那就是,软件是否应该成为我们核心业务的一部分呢?

这并不是一个容易回答的问题,本章将对该问题进行探究。在以下条件下,你可能想使软件成为核心业务的一部分:

1. 你所卖的软件依赖于你的专有软件。
 2. 当前,你的软件给贵公司带来了强悍的竞争优势。
 3. 贵公司的软件开发和维护效率远胜过你的竞争对手。
- 如果外包与你的核心业务具有如下关系,那么你也应该好好考虑一下软件外包:
1. 软件主要是用于企业运营的,而不是一个产品。
 2. 与你的竞争对手相比,你的软件没有显著的优势。
 3. 你的开发和维护效率微不足道。

一旦你认为外包是软件运营的一部分,或是软件运营中特定的应用,并且外包也与你商业计划相匹配,那么在外包协议中,我们就需要考虑如下一些主题:

- ☐ 在谈判过程中,必须确定可交付的软件的规模,最好使用功能点。
- ☐ 应用程序的成本和进度估算必须是正规的、完整的。
- ☐ 在合同中,用户的需求蔓延必须以双方都满意的方式呈现。
- ☐ 在合同中,需要对独立的服务条款和进展情况进行某种形式的估算。
- ☐ 合同中应该包含预期的质量水平。
- ☐ 供应商必须使用有效的软件质量控制措施。
- ☐ 如果合同要求生产率和质量的提高都是基于一个初始基线,那么我们在创建基线的时候,必须非常小心,并且确保基线是准确的,同时对双方也是公平的。
- ☐ 在开发过程中,必须对进度以及存在的问题进行跟踪,并且跟踪必须是完备的,不能忽视或故意隐瞒存在的问题。

幸运的是,这八个主题都是比较容易控制的,如果可以的话,应该将它们理解为麻烦。一个有趣的迹象是,如果外包供应商使用最先进的质量控制方法,那么他们是能够处理大型应用的。

大型软件应用所使用的先进技术包括:复杂的缺陷预测方法、度量缺陷去除效率的方法、缺陷预防方法、正规的设计和代码审查、拥有软件质量保证(SQA)部门、使用测试专家以及使用各种质量相关的工具,如缺陷跟踪工具、复杂度分析工具、调试工具以及测试库控制工具等。

在软件外包合同中,另一个重要的最佳实践是处理需求变更,因为这种情况很常见。

对于软件开发合同，一种处理用户需求变更的有效方法是在合同中包含成本变更的比例。例如，假设一个合同的初步协议是，开发一个拥有 1000 个功能点的应用程序，每个功能点价值 1000 美元，那么该协议的总价值为 100 万美元。

该合同应该包含以下几种类型的需求变更，对每种新增需求，对应的成本也在不断地攀升：

初始的 1000 个功能点	=	1000 美元 / 功能点
在合同签订后 3 个月增加新功能	=	1100 美元 / 功能点
在合同签订后 6 个月增加新功能	=	1250 美元 / 功能点
在合同签订后 9 个月增加新功能	=	1500 美元 / 功能点
在合同签订后 12 个月增加新功能	=	1750 美元 / 功能点
根据用户的要求，将功能删除或者延期	=	250 美元 / 功能点

在软件维护和功能增强的外包协议中，也可以使用以下类似的条款，如：

正常的维护和缺陷修复	=	每年每功能点 250 美元
大型机到客户端 - 服务器的转换	=	每系统每功能点 500 美元
“大规模更新”特定的搜索和修复	=	每系统每功能点 75 美元

（请注意，在美国，每个功能点的实际成本相差很大，小型终端用户项目的功能点约为 300 美元，大型军事软件项目的功能点则可能超过了 5000 美元。这里显示的数据仅仅用来说明基本的观点，并不是实际合同中所使用的。）

在开发和维护合同中，使用功能点指标的优势是，功能是由用户需求决定的，不能由承包商单方面进行增删。

总的来说，在 1000 个功能点的范围内，成功的软件外包项目通常具有如下这些特点：

- ☐ 在需求阶段，每个月的需求变更小于 1%。
- ☐ 需求波动的总数不超过 1%。
- ☐ 每个功能点中不超过 5.0 个缺陷。
- ☐ 在开始测试之前，缺陷去除效率超过 65%。
- ☐ 在交付之前，缺陷去除效率超过 96%。

相反，在 1000 个功能点的范围内，失败的软件外包项目通常具有如下这些特点：

- ☐ 在需求阶段，每个月的需求变更大于 2%。
- ☐ 需求波动的总数超过 2%。
- ☐ 每个功能点中超过 6.0 个缺陷。
- ☐ 在开始测试之前，缺陷去除效率小于 35%。
- ☐ 在交付之前，缺陷去除效率小于 85%。

如果对项目已取消或失败的项目进行细致分析，我们会很容易分辨出失败项目与成功项目的显著特征。经验丰富的项目经理都知道，虚假的乐观估计、贫乏的需求变更计划、贫乏的质量控制方法以及掩耳盗铃式的进度跟踪往往会导致项目失败和灾难。相反，准确的估算、谨慎的变更控制、真实的进度跟踪以及严格的质量控制则是项目成功的基石。

另一个复杂的话题是，将那些工作外包出去之后，内部员工接下来该干什么。最佳实践是，他们在自己的公司被重新分配，并且由他们来决定不外包的软件应用和任务。但是，一般情况下是由外包公司来承担人事分配，而这也是根据公司的具体情况采取的一个很好的、很公平的做法。最坏的情况是，其工作外包的员工将被裁掉。

除了外包整个应用甚至是项目组合，在部分外包协议中也有专门的主题，如测试、静态分析、质量保证、技术写作等。然而，这部分的转让也可以在内部完成，由承包商在现场操作，所以很难因为这些具体的专题而将外包服务从合同中剥离。

外包是一个重要的商业决定。对于供应商和客户之间的合同，应该采用最佳的做法，即增加成功的可能性，最大限度地减少诉讼的可能性。

在一般情况下，维护外包协议是不会出现麻烦的，不太可能像开发外包协议一样最终出现诉讼。事实上，如果应用的维护外包出去，那么就可以腾出足够的人员来处理积压的应用了，并且将精力集中在新开发的应用上。

随着经济的不断恶化，外包今后将存在很大的不确定性。软件依然是一个重要的商品，因此毫无疑问，外包依旧是一个重要的产业。然而，经济危机、通货膨胀率以及货币价值的变化将改变离岸外包的平衡，使得外包从一个国家转移到另一个国家。事实上，如果全世界都发生通货紧缩，那么美国也可能会扩大自己的外包范围。

2.8 使用承包商和管理顾问

2009年，美国大约有10%~12%的软件人员不是公司的全职雇员。他们是承包商或顾问。

在任何指定的工作日，在任何指定的500强公司中，大约都会有10个管理顾问会与高管和经理一起讨论一些专门的主题，如基准、基线、战略规划、竞争分析等主题。

这两种情况都可以被看作有益的实践，并且往往也属于最佳实践的范畴。

所有的公司都有软件开发的高峰和低谷。如果全职技术人员的工作负载一直是在高峰，那么他们在从事任何工作的时候也都不会出现低谷期。相反，如果全职技术人员的工作负载一直是在低谷期，那么当有重要的新任务时，公司将会出现技术人员短缺的情况。

最佳的做法是让员工的年平均工作负载适中。那么，当项目需要额外的资源时，引入的承包商要么是为了自己的新项目，要么是接管如维护一样的标准活动，从而为公司腾出更多的技术人员。换句话说，让5%~10%的全职人员的工作量低于峰值需求是具有成本效益的战略。

引入管理顾问主要是获得特殊技能和知识，而这些知识和技能可能在内部不容易得到。以下列举了一些主题范例，这些主题是管理顾问所具有的技能，而专职工作人员往往则缺乏这些技能：

- 比较基准与行业规范的差异。
- 在开始改进流程之前设定基线。
- 教授新的或有用的技术，如敏捷开发、六西格玛以及其他的技术。

- 度量和指标，如功能点分析。
- 选择国际供应商。
- 新产品的战略和营销规划。
- 准备诉讼或诉讼辩护。
- 协助改进公司的流程。
- 试图拯救有问题的项目。
- 为公司的上市、兼并、收购以及企业融资提供建议。

管理顾问那里是一个有用的渠道，他们可以提供同类公司的专门研究信息。管理顾问为公司付出的是专业知识，而不是工作时间，并且许多成功的管理顾问都是他们所在领域的顶尖专家。

管理顾问有战略和战术两方面的作用。他们的战略工作涉及的范围很广泛，如市场地位和软件组织结构的优化。他们的战术作用主要是在以下领域：如六西格玛、启动测量方案以及协助收集功能点数据等。

一般情况下，在软件开发和维护上，使用按小时付费的承包商；而在具体的专题上，则使用管理顾问，这对很多大型企业和政府机构都有好处。如果不经常采用最佳实践，那么使用承包商和管理顾问至少也是最佳的实践。

2.9 选择软件方法、工具以及做法的最佳实践

遗憾的是，在软件行业中，很少出现精心挑选的方法、工具以及实践。要么使用现成的方法开发应用，要么急于采用最新的方法开发应用，如 CASE、I-CASE、RAD 以及敏捷开发等。

选择软件开发方法的明智之举是，开始的时候，采用各种方法审查应用程序的基准，然后针对软件项目的具体规模和类型，选择能获得最佳结果的方法。

在编写本书的时候，成千上万的基准是由非营利性的国际软件基准组织（ISBSG）所提供的，这些基准都是用最常用的方法所表示的。其他机构提供的基准也可以使用，如软件生产力研究所以及 David 咨询集团等。然而，ISBSG 提供的基准免费向公众开放，并且访问也最方便。

当前可选择的软件开发方法有（按英文字母顺序排列）：敏捷开发、净室开发、水晶开发（Crystal Development）、动态系统开发方法（DSDM）、极限编程（XP）、混合开发、迭代开发、面向对象开发、基于模式的开发、个体软件过程（PSP）、快速应用开发（RAD）、Rational 统一过程（RUP）、螺旋式开发、结构化开发、团队软件过程（TSP）、V-模型开发以及瀑布模型等。

除了前面提到的，还有一部分的开发方法只针对特定的阶段或活动。这部分方法包括（按英文字母顺序排列）：代码审查、数据状态设计、设计审查、基于流的编程、联合应用设计（JAD）、精益六西格玛、结对编程、质量功能展开（QFD）、需求审查以及软件六西格玛等。虽然这部分方法不是全程的开发方法，但是我们可以用来衡量它们对质量和生产率的

影响。

在选择方法和实践的时候，先罗列一个主题清单，然后对它们进行评估是很有必要的。要评估的主题包括：

应用程序规模的适用性 对于拥有 10 到 10 万个功能点的应用，使用这些方法的效果如何。对于较小的应用，敏捷开发似乎有很好的效果，而团队软件过程（TSP）则似乎更适合大型系统，正如 Rational 统一过程（RUP）一样。当然，混合方法也需要包括在内。

应用程序类型的适用性 对于嵌入式软件、系统软件、Web 应用、信息技术应用、商业应用、军用软件以及游戏等，使用这些方法的效果如何。

应用程序性质的适用性 对于新开发、功能增强、保修期内的修复以及遗留应用的改造等，使用这些方法的效果如何。有几十种的开发方法，但很少有方法还包括维护和功能增强。截至 2009 年，大多数“新”的应用软件都已经替代了日益老化的遗留应用。因此对于隐藏的需求、功能增强以及改造来说，数据挖掘应该是软件方法学中的标准功能。

应用程序属性的适用性 这些方法支持软件应用的重要属性，包括但不限于以下这些方面：缺陷预防、缺陷去除效率、最大限度地减少安全漏洞、实现最佳的性能以及实现最佳的用户交互。如果一个开发方法不包括质量控制和质量度量，那么这个开发方法就真的不适合关键的软件应用。

应用程序活动的适用性 如果该方法支持需求、架构、设计、代码开发、可重用性、预备审查、静态分析、测试、配置控制、质量保证、用户信息、后续维护、改进以及客户支持等，其效果如何。

最基本的是，我们应该慎重地选择能满足项目具体需求的方法，而非仅仅因为它们是当前流行的方法，或因为没人知道其他的方法，我们就使用这些方法。

在编写这本书的时候，有正规的技术选择的软件应用似乎不到 10%。人们在使用方法的时候，一般情况下都没有考虑方法是否能够很好地匹配应用程序。大约有 60% 的人采用当地常用的方法，约有 30% 的人采用最近流行的方法，如敏捷开发。

开发过程是由一系列标准的活动组成的，并且通过执行这些活动以建立一个软件应用。（开发过程和开发方法基本上是同义词。）

对于常规的软件开发项目来说，工作分解结构（Work Breakdown Structure, WBS）中通常包含 25 项活动和 150 项任务。而对于敏捷开发来说，工作分解结构中通常大约包含 15 项活动和 75 项任务。

大型系统的工作分解结构取决于：（1）软件应用是否要从头开始开发；（2）软件应用是否涉及修改程序包或修改遗留应用程序。在当前，对传统项目进行修改实际上超过了新项目的开发。

在额定规模为 10 000 个功能点的项目中应该包括收购和修改商业套装软件，一个项目的有效开发过程将类似于以下所罗列的：

1. 需求收集
2. 需求分析
3. 需求审查

4. 对已有的同类应用进行数据挖掘，以提取业务规则
5. 架构
6. 外部设计
7. 内部设计
8. 设计审查
9. 安全漏洞分析 (Security Vulnerability Analysis)
10. 正规的风险分析
11. 正规的价值分析
12. 商用现货^② (Commercial-off-the-shelf, COTS) 软件包的分析
13. 需求 / 软件包映射
14. 联系软件包用户协会
15. 软件包的授权及收购
16. 在选定软件包之后，对开发团队进行培训
17. 修改软件包的设计
18. 修改软件包的开发
19. 特定功能的开发
20. 收购认证的可重用材料
21. 审查软件包的修改
22. 修改软件包的文件
23. 审查文档和帮助文件
24. 修改软件包的静态分析
25. 修改软件包的通用测试
26. 修改软件包的专业测试包 (包括性能、安全性)
27. 修改软件包的质量保证审查
28. 软件包用户的人事培训
29. 培训客户支持和维护人员
30. 修改软件包的部署

这些高级别的活动通常是一个完整的工作分解结构，它拥有 150 至 1000 个任务和较低级别的活动。对于大型应用的使用手册来说，做一个完整的工作分解结构实在是太困难了。因此，先进的公司一般会经常使用项目管理工具，如 Artemis Views、Microsoft Project、Primavera 以及类似的工具。

由于需求每个月的变更率都在 2% 左右，因此这些活动必须通过以下的方式来执行：在开发过程中，开发团队要能较快适应变更；也就是说，对每一个重要的可交付成果来说，某种形式的迭代开发是必需的。

② 商用现货 (Commercial Off-The-Shelf, COTS) 指的是那些可以很容易获得的现成产品，这一术语有时也在军事采购规则中。——译者注

然而, 由于在合同中设置的交付时间表是固定的, 因此需要强制性地规定大型应用应该开发多个版本。对于某个确定的点, 初始版本中其他所有功能都应该被冻结, 并且这个点必须添加到后续发布的版本中。这将迭代开发的概念扩展成了一种跨年、跨版本的理念。

许多顶尖的公司(如 IBM 和 AT & T)很早就认识到变更将一直伴随着软件应用。这些公司往往有固定的发布时间间隔, 并且, 正式的版本规划至少覆盖当前版本的两个后续版本。

正规的风险分析和价值分析也是软件复杂性的指标。如在诉讼中所提到的, 失败的项目往往不进行风险分析, 所以他们往往对导致延期或成本超支的因素感到惊讶。

先进的企业经常对重大课题进行正规的风险分析, 这些课题包括: 人员流失的可能性、需求变更、质量以及其他关键主题。然而, 有一种形式的风险分析所有公司都做得不是很好, 即使是最先进的公司也不例外, 这就是安全漏洞。即使我们真的做了安全性分析, 那往往也是徒劳的。

通过对大型软件工程项目的跟踪记录, 我们已经证实了有很多方法。这些方法包括: 软件工程研究所创建的 CMM(软件能力成熟度模型)以及 Watts Humphrey 和 SEI 所创建的新型团队软件过程(TSP)和个体软件过程(PSP)。Rational 统一过程(RUP)在大型软件项目上也取得了一些不错的成绩。对于较小的应用程序来说, 各种风格迥异的敏捷开发和极限编程(XP)有很好的效果。其他的方法对大型应用也有一定的价值, 如面向对象开发、模式匹配、六西格玛、正式审查、原型以及重用等。

除了以上所提到的“纯”方法, 如团队软件过程(TSP), 混合方法也是相当不错的。混合方法将几种不同的方法混合在一起使用, 以满足特定项目的需求。截至 2009 年, 混合式开发方法似乎比纯方法有更广阔的使用范围。

总体来说, 混合使用六西格玛、能力成熟度模型、敏捷开发以及其他一些方法有一些显著的优点。原因是, 这些“纯”方法所适应的项目规模和类型是相当狭窄的, 然而它们却是最有效的。方法组合显得更加灵活, 可以匹配任何规模、任何类型的软件项目。然而, 将各种方法混合在一起需要一定的专业知识, 并且得确保混合方法是最佳组合。这是一个专家的工作, 而不是一个新手的工作。

在美国各主要城市, 都有许多软件过程改进网络(SPIN)分会。这些组织有多如牛毛的会议, 他们对传播信息起到了关键的作用, 这些信息涵盖各种方法和工具的优劣。

显而易见, 选择任何方法的时候, 都应该提供对以前方法的改进。例如, 目前美国软件的平均缺陷总数约为 5.00 个/功能点, 平均缺陷去除效率约为 85%, 因此交付的缺陷数约为 0.75 个/功能点。

任何新应用应该降低潜在缺陷, 提高缺陷去除效率, 并降低交付的缺陷个数。一种改进方法的建议值可以是: 3.00 个缺陷/功能点, 95% 的去除效率, 并且交付的缺陷不超过 0.15 个/功能点。

此外, 任何真正有效的过程应该以提高生产率为目的, 并且增加认证的可重用材料的数量, 以用于软件构建。

2.10 认证方法、工具以及实践

软件行业往往是从一个时尚转向另一个时尚，并且每种方法都曾夸下海口，说可以使软件的生产率和质量达到一个新高度。对软件产业来说，最有价值的是非营利性的组织，它们会对方法、工具以及实践的有效性进行客观的评估。

标准化度量实践也是很有用的，它们收集所有大型软件项目生产率和质量方面的数据。

这项工作并不是那么容易做的。评估小组尝试使用每一种开发方法显然是不现实的，因为这样的方法在实践中可能会持续数年，并且还有数十种方法。

最为行之有效的方法是直接分析各个项目中的实证结果，这些项目综合运用了各种方法、工具和实践。而这些项目相关的数据可以从基准源获取，如国际软件基准组织 (ISBSG)；或从其他源头获取，如芬兰的软件度量协会。

要做到这一点，需要两种分类法：（1）软件应用分类法，通过项目的规模和类型来提供一个评估方法的结构，（2）软件方法和工具本身的分类法。

第三种分类法是软件的功能集，这种分类法也是有益的；但是截至 2009 年，并没有足够的细节来证明该方法也是有益的。三种分类法的基本思路都是模式匹配。换句话说，应用以及其对应的功能集和开发方法都涉及共同的问题，如果能够获知与模式相关的问题，那么这三种分类法也是很有用的。这将推动软件行业逐步向着构建标准可重用组件的方向前进。

当前这两种分类法涉及两个问题，即什么样的软件可以使用这种方法，以及这种方法自身包含哪些特征。

将拥有超过 1 万个功能点的大项目与少于 1000 个功能点的小项目进行比较，这个结果显然是不公平的。同样，将嵌入式军事应用与 Web 项目进行比较，也是不公平的。因此，对软件项目按标准进行分类是评估和选择方法的前提。

多年来，笔者和同事们一直在进行评估和基准的研究，在研究中，我们发现一种四层分类法貌似为软件应用提供了一个合适的结构：

属性 属性这一术语是指该项目是属于一种新开发，还是一种改进，还是一种翻新，或者是其他什么东西。属性参数的例子包括新开发、传统软件的功能增强、缺陷修复以及平台转换。

范围 范围这一术语用来确定项目的规模，从一个模块到一个企业资源规划 (ERP) 软件包。功能点指标以及源代码指标来表示应用的规模。规模范围的跨度很广，从少于 1 个功能点到超过 10 万个功能点都适应。为了简化分析，应用的规模可以是离散的，即 1 个功能点、10 个功能点、100 个功能点、1000 个功能点、10 000 个功能点以及 10 万个功能点。范围参数的例子包括原型、演化原型、模块、可重用模块、组件、独立的程序、系统以及企业系统等。

分类 分类这一术语用来确定该项目是否供开发组织以外的其他组织使用，或者是否在网上或以其他形式对外发布。分类参数的例子包括单个位置的内部应用、多个位置的内部应用、公共领域中的外部应用（开放源码）、商业化的外部应用、Web 上的外部应用以

及硬件设备中嵌入的外部应用。

类型 类型这一术语用来确定应用程序属于那种类型的软件应用，如嵌入式软件、信息技术、军事应用、专家系统、电信应用、计算机游戏或者其他类型的应用。类型参数的例子包括批处理应用、交互式应用、Web 应用、专家系统、机器人应用、过程控制应用、软件科学、神经网络的应用以及同时包含多种类型的混合应用。

这四个部分的分类法可用于定义和比较软件项目，以确保相似的应用之间可以进行比较。有趣的是，当使用功能点指标进行度量时，假如应用程序共享这种相同的分类模式，那么它们往往具有相同的规模。

第二种分类法将定义开发方法本身的功能。它包含 25 个主题：

为软件方法分析提供分类法

1. 团队组织
2. 团队成员的专业化
3. 项目管理——规划和估算
4. 项目管理——跟踪和控制
5. 变更控制
6. 架构
7. 业务分析
8. 需求
9. 设计
10. 可重用性
11. 代码开发
12. 配置控制
13. 质量保证
14. 审查
15. 静态分析
16. 测试
17. 安全
18. 性能
19. 大型应用的部署和定制
20. 文档和培训
21. 本地化
22. 客户支持
23. 维护（缺陷修复）
24. 功能增强（新功能）
25. 改造

这 25 个主题只是软件生命周期中的一部分，包括最初的需求启动以及最终的应用退役。在评估方法的时候，此检查清单可以用来显示部分的时间安排，以及该方法支持的主题。

例如敏捷开发涉及以上 25 个因素中的 8 个：

1. 团队组织
2. 项目管理——规划和估算
3. 变更控制
4. 需求
5. 设计
6. 代码开发
7. 配置控制
8. 测试

换句话说，敏捷主要用于新应用开发，而非维护和改进遗留应用。

团队软件过程涉及以上 25 个因素中的 16 个：

1. 团队组织
2. 团队成员的专业化
3. 项目管理——规划和估算
4. 项目管理——跟踪和控制
5. 变更控制
6. 需求
7. 设计
8. 可重用性
9. 代码开发
10. 配置控制
11. 质量保证
12. 审查
13. 静态分析
14. 测试
15. 安全
16. 文档和培训

TSP 也是很重要的开发方法，但其主要面向软件质量控制，也包括规划和评估项目管理组件。

评估软件的方法、工具以及实践的另一个重要方面是度量所得到的生产率和质量水平。度量是软件行业的一个薄弱环节。要想有效地评估方法和工具，我们就必须小心谨慎，切莫操之过急。功能点度量最适用于测评和经济方面的。应当尽量避免不稳定的有害指标，如代码行和平均缺陷成本。

但是，为了确保使用特殊方法的项目能进行横向比较，度量措施应该逐渐下降到特定的活动水平。如果只是用项目或阶段级别的数据，那么用于评估的措施将是不准确的。

虽然并不是每个项目都使用每个活动，但是笔者在活动的级别上使用广义的活动成本图表，以收集基准数据。这些活动包括：

活动级软件基准报告列表

1. 需求 (初始)
2. 需求 (更改和添加)
3. 团队教育
4. 原型
5. 架构
6. 项目规划
7. 初步设计
8. 详细设计
9. 设计审查
10. 编码
11. 可重用材料的收购
12. 软件包收购
13. 代码审查
14. 静态分析
15. 独立核查和验证
16. 配置控制
17. 集成
18. 用户文档
19. 单元测试
20. 功能测试
21. 回归测试
22. 集成测试
23. 性能测试
24. 安全性测试
25. 系统测试
26. 现场测试
27. 软件质量保证
28. 安装
29. 用户培训
30. 项目管理

除非能确定具体的活动, 不然, 本质上无法将项目与项目进行有效的比较。

另外, 也需要对软件质量进行评估。然而, 应该优先考虑质量成本 (COQ), 并且应该始终包括两个重要的补充措施, 即潜在缺陷和缺陷去除效率。

软件应用的潜在缺陷数是在需求、设计、代码、用户文档以及错误修复中所发现的错误总和。缺陷去除效率是在软件交付之前所发现的缺陷的百分比。

截至 2009 年, 潜在缺陷的平均值如下:

在交付之前,累积缺陷去除效率大约只有 85%。因此在评估方法的时候,应该从它们减少了多少潜在缺陷,以及增加了多少缺陷去除效率的层面上去考虑。如像团队软件过程(TSP)一样的方法将缺陷降低到了 3.0 个/功能点,并且将缺陷去除效率提升到了 95% 以上,因此 TSP 通常被视为最佳的实践方法。

缺陷来源	每个功能点中缺陷的个数
需求缺陷	1.00
设计缺陷	1.25
代码缺陷	1.75
文档缺陷	0.60
不良缺陷修复(次生缺陷)	0.40
总计	5.00

除此之外,也需要对生产力进行评估。笔者所使用的方法是选择平均值或中点的方法,比如以能力成熟度模型集成(CMMI)的 1 级为基准。例如,规模为 10 000 功能点的 CMMI 1 级应用,每名员工的月平均生产力约为 3 个功能点/月。替代方法改进了这些结果,并且可以和初始值进行比较,这些替代方法包括:团队软件过程(TSP)和 Rational 统一过程(RUP)。当然,有些方法也可能会降低生产率。

最基本的是,评估(截至 2009 年)几乎没有被使用过的软件方法、工具以及实践。结合已完成的项目中的项目活动级基准数据,可以使用一种正规的分类法来确定特定类型的软件应用,而使用另一种正规的分类法来识别方法的特性。准确的质量数据也是不可或缺的,它涉及潜在缺陷和缺陷去除效率水平两方面。

认证方面另外一个重要的主题是,显示改进后的结果与美国当前平均水平的差异。由于平均值是随着应用的规模和类型而变化的,因此滑准计算法^④(Sliding Scale)是必需的。例如,目前从需求到交付的平均日程安排可以被近似提高到应用程序功能点总数的 0.4 次幂。在理想的情况下,最佳的开发过程会将指数减少到功能点总数的 0.3 次幂。

在美国,软件应用目前的缺陷去除效率只有 85% 左右,而一种改进的开发过程则可以使缺陷去除效率超过 95%。

在开发过程中可能会遇到这种情况,即潜在缺陷或错误总数会被提高到应用功能点总数的 1.2 次幂,并且大系统的缺陷个数更是高得令人难以想象。一种改进的开发过程可以将潜在缺陷降低到总数的 1.1 次幂。

当前软件应用中认证的可重用材料数量少于 50%,并且平均使用率约为 25%。如果使用认证可重用材料的数量超过 85%,并且较为常见的应用类型中的使用量达到 95% 以上,那么软件产业势必会有更好的经济形势。

最根本的是,认证需要查看量化结果,并且包含使用新方法后的效果。认证的另一个方面是,查看来自国际软件基准组织(ISBSG)的可用报告和基准。由于 ISBSG 收集的历史基准已经达到了 5000 多个项目,因此越来越多的方法可以通过充足的细节来表示,并且还可以进行多元回归研究,同时也可以评估方法的影响。

④ 滑准计算法是用来计算滑准税(Sliding Duties)的,它是对进口税则中的同一种商品按其市场价格标准分别制订不同价格档次的税率而征收的一种进口关税。其高档商品价格的税率低或不征税,低档商品价格的税率高。——译者注

2.11 软件应用的需求

截至 2009 年,假如从应用是否是首次开发的层面来讲,80% 以上的软件应用都不是新的。当前,大多数应用都是旧的或过时的应用的替代品。

由于这些应用程序都是过时的,因此它们的书面功能规格说明文档通常会被忽视,并且这些规格说明也是过时的。然而,虽然缺乏当前的文档,但是旧应用却包含数百或数千的业务规则和算法,因此需要将这些业务规则和算法转换到新应用中。

因此,截至 2009 年,需求分析不应该只涉及新的需求,而且还应包括对遗留代码进行数据挖掘,以提取隐藏的业务规则和算法。有些工具可以做到这一点,也有很多的维护工作平台可以显示代码,并且帮助提取潜藏的业务规则。

虽然清晰的需求是一个值得称赞的目标,但是对于拥有 10 000 个功能点的软件应用来说,这个目标只能是个奢望。到目前为止,笔者只观察到了一个小项目,它的功能点少于 500 个,并且该应用的初始需求是清晰的和不变的。

对于大型应用来说,业务需求是动态的,并且不可能是一成不变的。许多外部事件都会使软件应用的需求发生变化,如税法的变化、企业结构的变化、业务流程的再造以及兼并和收购等。另外,大型应用的开发一般需要几年的时间,这使得情况变得更加复杂了。一个公司仅仅为了满足一个软件项目的需求而冻结其所有的业务规则,显然是不现实的。

最典型的情况是处理拥有 10 000 个功能点应用的需求,收集和分析初始的需求将花费数个月。在随后的设计过程中,每个月的新需求和变更需求将达到 2% 左右。最终的需求总量将会达到初始需求的 50%。在发布了软件应用的第一个版本之后,应该终止这些新的和变更的需求,并且在 9 ~ 12 个月之后,在后续的版本中添加新的需求和变更的需求。

对于拥有 10 000 个功能点的项目来说,收集和分析需求的先进技术包括:

- 使用联合应用设计(JAD)来收集最初的需求。
- 对于质量需求,使用质量功能展开(QFD)。
- 对于安全性分析和安全漏洞预防,使用安全专家。
- 对于新应用的主要功能,使用原型开发。
- 通过挖掘遗留应用,以获取新项目的需求和业务规则。
- 让用户参与敏捷项目开发。
- 确保需求被清楚地表示,并且很容易理解。
- 用户和供应商双方都使用正规的需求审查。
- 创建一个客户/供应商联合变更控制委员会。
- 为特定功能的变更选择对应领域里的专家。
- 确保需求的可跟踪性。
- 需求变更的多版本分割。
- 使用自动化需求分析工具。
- 仔细分析软件包的功能。

对于拥有 10 000 个功能点的项目来说,每个月的需求变更比例稍低于 0.5%,累计增量

不超过原始需求的 10%。然而，最大的增量可以达到 200%。在设计和编码阶段，每个月需求变更的平均比例在 1% ~ 3% 之间，而之后的变更则被添加到了以后的版本中了。

同时使用 JAD 会议、仔细的需求分析、需求审查以及原型可以使需求过程在技术和管理的控制之下。

虽然有时需要数月甚至数年才能看到项目的结果，但是大型软件项目的成败在需求阶段就已经一目了然了。成功的项目在收集和分析需求上，比失败的项目更完整、更彻底。因此，成功的项目变更很少，以及需求蔓延也很少。

然而，由于大多数新应用都是遗留应用的翻新，因此需求应该包括数据挖掘，以提取遗留应用的潜在业务规则和算法。

2.12 用户参与软件项目

假如我们要设计和构建有 10 000 个功能点的应用程序，但却不了解用户的需求，这无异于白日做梦。此外，当应用程序在开发中时，用户通常会参与评审，并协助审核具体的项目可交付物，如目录和文档。对于任何关键的输入、输出以及功能，用户也可以审核，甚至参与原型的开发。用户参与是新型敏捷开发的一大特色，用户代表会加入到项目团队中。对于任何重大的应用，最先进的用户参与包括：

1. 联合应用设计 (JAD) 会议
2. 质量功能展开 (QFD)
3. 回顾从遗留应用中挖掘的业务规则和算法
4. 一个全职基础上的敏捷项目
5. 需求评审
6. 变更控制委员会
7. 评审由承包商提供的文件
8. 设计评审
9. 使用由承包商提供的原型以及界面截屏的示例
10. 通过培训课程来学习新应用
11. 从设计阶段一直到测试阶段的缺陷报告
12. 验收测试

用户参与是比较费时的，但却很有价值。平均而言，用户的贡献量是软件技术团队的 20%。用户参与的范围可以低于 5%，但是也可以高于 50%。然而，对于复杂的大型项目来说，如果用户的参与度不到开发团队所付出努力的 10%，那么该项目在最终完成时，将在用户满意度上存在一定的风险。

敏捷开发中会有一个全职用户代表，该用户作为项目团队的一部分。这种方法确实很适合小项目以及小规模的用户。但是当用户数量变得很庞大的时候，这种方法就显得捉襟见肘了，如拥有数以百万计用户的 Microsoft Office 或 Microsoft Vista。对于拥有数以百万计用户的应用来说，没有一个用户能够理解整个范围内所有可能的用途。

对于这些高使用率的应用来说,可以调查数以百计的用户或焦点小组,并且将几十个用户提供的意见作为首选。此外,易用性实验室也是很有帮助的,用户可以尝试使用功能和原型。

正如我们所看到的,对于拥有1个功能点、10个功能点、100个功能点、1000个功能点、10 000个功能点以及10万个功能点的应用来说,不存在“一刀切”的方法。各种规模的应用以及各种类型的软件都需要自己独特的最佳方法和实践。

医学上的情况也是如此。没有任何抗生素或治疗剂能从容应对所有的疾病,如细菌性疾病和病毒性疾病。每种情况都需要一个独特的配方。此外,和医学上一样,某些情况可能已经到了病入膏肓的境地,即使是华佗再世,也回天乏力。

2.13 软件应用中的行政管理支持

新应用的行政管理支持主题是由应用的整体规模所确定的。高管们对500个功能点以下项目的参与度可能为零,因为这些项目的成本是如此之低,风险也是如此之低,以至于高管们根本不感兴趣。

然而,对于拥有10 000个功能点的大型应用来说,行政监督则是再正常不过的了。大型软件项目频繁的失败已经引起企业高管对软件经理极大的不信任。事实上,由于高失败率、经常超支以及平庸的质量水平,大公司的软件组织已经被视为公司最头疼的组织。

就总体而言,软件行业中最先进的行政管理支持主要有以下几个角色:

- 核定软件项目的投资回报率(ROI)。
- 为软件开发项目提供资金。
- 在监督、治理以及项目总监的角色上指派关键的管理人员。
- 审查里程碑、成本以及风险状况报告。
- 确定项目超支或延期后,是否会使项目的投资回报率低于企业的目标。

即使管理人员承担了所有常见的角色,项目仍然可能出现问题和故障。软件项目失败的关键是,如果别人提供了假情报,而不是准确的状态报告,那么高管很可能没法做出正确的商业决定。如果软件项目的状态报告和风险评估轻轻略过了项目中存在的问题和技术难点,那么管理人员将很难像他们期待的那样控制项目。因此,不充分的报告和不坦诚的风险评估将推迟最终稳健的行政决策的出现,该决策试图通过终止失控的项目以进一步节省开支。

确定项目失控的原因是一个正常公司的行政责任。许多大公司的高管都不信任软件的原因是,软件项目都有失控的倾向,并且往往没法提供准确的状态报告。因此,高层管理人员(如高级副总裁、首席运营官、首席执行官)常常会认为软件是相当令人沮丧的,并且和其他运营单位的产品相比,软件也显得很专业。

笔者曾经遇到了一些公司的高管,他们都认为软件组织是最不值得信赖的企业组织。不幸的是,在软件项目经历了那么多的延期和成本超支后,企业高管似乎对软件经理更加不信任了。

在软件产业中的每个人都有有一个共同责任，那就是将软件项目经理和软件工程师的专业能力提高到一个水平，并且使我们受到企业客户主管的信赖。

2.14 软件架构和设计

对于拥有 1000 个功能点的小型独立应用，架构和设计往往是非正式的活动。然而，当应用的规模增加到 10000 甚至 10 万个功能点的时候，架构和设计就变得越来越重要了，并且也变得越来越复杂，越来越昂贵了。

企业架构的范围相当广泛，它试图将企业的项目组合与对企业的总业务需求匹配，这些业务需求包括：销售、市场营销、金融、制造、研发以及其他运营单位。目前，规模最大的企业架构涉及 5000 多个应用，10 万多个功能点。

一个大型软件应用的架构涉及应用的整体结构，并且还涉及该应用与其他应用以及外界之间的连接特性。截至 2009 年，有许多架构可供使用，但是新应用需要选择一个特定的架构。这些架构包括单一应用架构、面向服务的架构（SOA）、事件驱动架构（Event-Driven Architecture）、对等架构、管道与过滤器模式（Pipes And Filters）^⑤、CS 架构、分布式架构以及专门为国防和政府应用设计的架构等。

一位名叫 John Zachman 的 IBM 同事开发了一个有趣的并且也很有用的架构，该架构说明了一些需要包括在大型软件应用架构决策中的主题。在表 2-2 中显示了一个完整的 Zachman 架构（Zachman Schema）。

表 2-2 Zachman 架构范例

	数据	功能	网络	人	时间	动机
规划者						
拥有者						
设计者						
构建者						
承包商						
企业						

在 Zachman 架构中，每列显示了基本的活动，每行显示了参与软件的基本人员。行和列的交点详细罗列了该点的任务和决定。表 2-2 揭示了数量相当多的变量，这些变量都是需要处理的，只有处理了这些变量才能为大型软件应用开发架构。

软件应用的设计与架构相关，但是也涉及许多其他的因素。截至 2009 年，设计方法的选择是悬而未决的，并且有 40 多种可供选择的设计方法。统一建模语言（UML）和用例是当前最热门的设计方法，但也有人在用其他的一些设计方法。其他的设计方法包括老式的

⑤ Pipes and Filters 模式为系统现有组件提供了一种处理数据流（过滤）及将传送数据的相邻组件（管道）连接起来的架构。这种架构提供了可重用性和可维护性，并将系统进程中不同的、易识别的、独立但有些复杂的任务分离。——译者注

流程图、HIPO图^①、Warnier-Orr图^②、Jackson图^③、Nassi-Schneiderman流程图^④、实体关系(E-R)图、状态转换图、动作图、决策表、数据流图、面向对象的设计、基于模式的设计以及其他一些混合的方法等。

大量的软件设计方法和图表技术说明了一个问题,即目前还没有出现一个单一的最佳实践。软件设计的基本主题包括:(1)给用户提供有关功能和特性的描述;(2)告诉用户如何使用它们。在内部设计的层面上,必须说明如何在内部链接这些功能和特性,并且说明如何共享这些信息。软件设计的关键要素包括安全方法和性能问题。此外,在开发的时候,必须讨论其他的应用如何给该应用提供数据,以及其他应用如何从该应用获取数据。显然,设计还必须解决硬件平台和软件平台的问题,如在特定操作系统下如何使用应用程序。

由于许多软件应用是非常相似的,因此可以将通用应用的基本特性、功能以及结构元素记录到模式中,并且重复使用。一旦描述这些模式的标准方法从有竞争性的设计语言和图形方法中诞生,那么可重用的设计模式将会成为一个最佳实践。

通过使用自动工具来检查现有应用的结构,同时将一些架构模式可视化是可行的。事实上,挖掘现有软件的业务规则、算法以及架构信息是创建软件功能库的第一步,该库包括软件功能的可重用组件以及可行的分类。

企业架构也适合自身的模式分析。任何访问过同行业大量公司的顾问都不得不注意,软件项目组合与所有的保险公司、银行、制造业以及药品业等都有80%的相似度。事实上,新西兰政府要求所有的银行都使用相同的软件,这样做的部分原因是,可以使监管机构更容易监督审计和安全控制(虽然可能增加恶意软件以及拒绝服务攻击的风险)。

就像2009年,行业最需要的是有效的方法,这些方法可以将架构模式变得可视化,并且易于使用。信息的被动显示是不够的。还有一个更深层的需求是,将成本、价值、用户数量、战略方向以及其他类型的商业信息与架构联系起来。此外,有必要说明的是软件应用所使用的数据;从一个操作单元流动到另一个操作单元,从一个系统流动到另一个系统的信息以及数据流;即动态模型将是最好的表现形式。鉴于信息是复杂多样的,今后最有效的模式可视化方法可能是动态全息影像。

2.15 软件项目规划

在大型企业中,大型软件项目的规划往往会涉及规划专家以及自动化规划工具。截至2009年,最先进的软件项目规划技术已经适应于有10 000个功能点的大项目了。

① HIPO图(hierarchy plus input-process-output)是IBM公司于20世纪70年代中期在层次结构图(structure chart)的基础上推出的一种描述系统结构和模块内部处理功能的工具(技术)。——译者注

② Warnier图又称为Warnier-Orr图,可以表示数据结构和程序结构。——译者注

③ Jackson图是结构化设计的一种方法,Jackson图既可以表示程序结构,也可以表示数据结构,有利于结构化技术的实现。——译者注

④ Nassi-Schneiderman流程图(即N-S图)是由美国学者I.Nassi和B.Shneiderman提出的一种新的流程图形式,这种流程图完全去掉了流程线,算法的每一步都用一个矩形框来描述,把一个个矩形框按执行的次序连接起来就是一个完整的算法描述。——译者注

- 开发完整的工作分解结构。
- 收集和分析相似项目的历史基准数据。
- 由正规的项目办公室提供规划援助。
- 在项目开发过程中，考虑到员工的雇用和营业额。
- 使用自动化的规划工具，如 Artemis Views 或 Microsoft Project。
- 收集和分析需求的时候，考虑时间因素。
- 处理需求变更的时候，考虑时间因素。
- 如果需求蔓延很极端，需要考虑多个版本。
- 如果使用外包，需要考虑给定的传输软件。
- 如果涉及多个公司，需要考虑供应链。
- 对于一套完整的质量控制活动，需要考虑时间因素。
- 对可能发生的重大问题进行分析。

成功项目的规划确实做得非常好。然而，延期或已取消的项目几乎都有规划上的失误。最常见的规划失误包括：（1）不能有效地处理需求变更；（2）在项目期间，不能很好地预测员工的雇用和营业额；（3）没有分配足够的时间进行详细的需求分析；（4）没有分配足够的时间进行正规的审查、测试以及缺陷修复；（5）基本上忽略了项目可能存在的风险。

在先进企业里，大型项目通常都会由项目办公室提供规划支持。项目办公室通常会有 6 至 10 个工作人员，并且这些人都有精良的装备，如规划工具、评估工具、基准数据、跟踪工具以及其他形式的数据分析工具，如统计加工。

由于项目规划工具和软件成本估算工具通常都由不同的供应商提供，虽然他们共享数据，但是规划和评估却是不同的主题。正如大多数管理人员所使用的工具那样，业务规划涉及整个项目所需要的网络活动和关键路径；业务评估涉及成本和资源预测以及质量预测。这两个业务是相关的，但却是不同的。这两种工具是相似的，但却是完全不相同的。

如果规划和评估是由相似项目的基准数据所支持的，那么规划和评估也都是可信的。因此，所有大型项目都应该包括公共来源的基准分析，这些公共来源包括国际软件基准组织（ISBSG）以及内部的基准。正如在诉讼过程中所提到的，软件产业的主要问题是，准确的规划和预算通常都被拒绝，并且被不切实际的规划和预算所代替，这些不切实际的规划和预算通常都依据业务需求而非团队能力。通常情况下，这些不切实际的需求来自于客户或高级管理人员，而非项目经理。然而，如果没有类似项目的经验数据，准确的项目规划和评估就变得相当困难了。在风险分析的研究中，这是一个微妙但却不被认可的风险因素。

2.16 软件项目的成本估算

对于拥有 1000 个功能点的小型应用来说，手动估算和自动估算的准确率不相上下。然而，当应用的规模增加到了 10 000 个功能点以后，自动估计还是相当准确的；但是手动估计就差很多了，手动估算往往会遗漏关键的活动，不能处理需求的变更，并且还时常低估测试和质量控制。当项目超过 10 000 个功能点后，自动估算工具依然是最佳实践，而手动

估计则是接近专业过失了。

估算拥有 10 000 个功能点的软件项目是一个重要的活动。大型系统当前最先进的估算技术包括：

- ❑ 对于大型可交付项目来说，估算规模的正规方法是基于功能点。
- ❑ 对于大型可交付项目来说，估算规模的候选方法是基于代码行。
- ❑ 对于大型可交付项目来说，第三种估算规模的方法是使用信息，如目录、报告等。
- ❑ 包含对可重用材料的估算。
- ❑ 如果涉及多个公司，在估计中应该包含供应链。
- ❑ 如果涉及国际或分散的团队，应该包含交通费用。
- ❑ 将项目的估算和相似项目的历史基准数据做比较。
- ❑ 训练有素的项目估算专家。
- ❑ 软件估算工具（CHECKPOINT、COCOMO、KnowledgePlan、Price-S、SEER、SLIM、SoftCost 等）。
- ❑ 在估算中包含新的和不断变更的需求。
- ❑ 包含质量估算以及进度和成本估算。
- ❑ 风险的预测和分析。
- ❑ 估算所有项目的管理任务。
- ❑ 估算规划、规范以及成本跟踪。
- ❑ 有充足的历史基准数据，以防止对估算做任意更改。

在软件文献中，在估算工具的指标与专家的手工估算之间有一些争议。然而，在美国，对于超过 1 万个功能点的项目来说，几乎没有任何专家参与，大部分专家都效力于商业软件估算公司。

这样做的原因是，在整个职业生涯中，项目经理可能只会涉及一到两个拥有 10 000 个功能点以上的大型系统。另一方面，估算公司一般从几十个大型应用中收集数据。

对于大型应用来说，手动估算最常见的失误是，由于他们缺乏一定的经验，因此他们会过于乐观。虽然通常可以很好地估算编码工作，但是手动估算往往会低估书面工作的工作量、测试工作的工作量以及需求变更的影响。对大型应用来说，即使手动估算是准确的，但是每隔几个星期都更新手动估算（如需求变更）的代价也是非常高昂的。

通过了解诉讼，我们得到了一个很无语的结果，那就是，有时候准确的估算竟然被驳回，甚至是拒绝，这是因为这些估算是正确的！客户或高管们拒绝了原先准确的估算，取而代之的是一个凭空捏造出来的估算。这是因为项目原先的估算预测项目需要更长的时间以及更高的成本，而这超出了客户所能承受的范围，所以他们不接受这样的估算。当出现这种情况时，超过 80% 的项目都会失败，并且有 99% 的项目会出现严重的成本超支以及项目延期。

这个问题的一个解决方案是，通过相似应用的历史基准数据来支持估算。这些数据都可以从国际软件基准组织（ISBSG）或从其他一些地方获得。一般情况下，大家都认为基准比估算更可信，因此通过历史基准来支持估算是值得推荐的最佳实践。这种方法存在一个

问题，那就是 1 万个功能点以上的历史基准几乎是凤毛麟角，而超过 10 万个功能点的基准则几乎不存在。

失败的项目往往会低估要完成的工作量。失败的项目自始至终一直忽略项目的质量评估。高估生产率是成本超支和项目延期的另一种常见原因。低估书面工作成本也是一个通病。

令人惊讶的是，在估算编码的进度和成本上，成功的项目和失败的项目是相似的。但在估算测试时间和成本时，失败的项目往往过于乐观。在开发过程中，失败的项目往往会忽略需求变更，这会使项目的规模显著增加。

由于估算比较复杂，因此训练有素的估算专家往往是少而精。这些专家总是使用一个或多个领先的商业软件估算工具，有时也使用专门的估算工具。大约有一半领先的客户经常使用商业软件估算工具，并且同时使用的数量可能多达 6 个。对于拥有 10 000 个功能点的大型系统，手动估算一直都是不准确的。

当假设发生变化时，使用标准模板的手动估算将很难修改。因此，当正在进行中的项目发生了重大变化时，他们往往落后于实际情况。笔者曾经对 1000 个功能点以上的项目进行过观察，之后发现，项目的手动估算往往是不完整的，并且项目团队也过于乐观。

对于超过 10 000 个功能点的大型项目来说，手动估算是测试、缺陷去除效率以及成本的福音。然而，对于整个项目来说，手动估算却是危险的。

对于大公司里的许多大型项目来说，项目办公室一般会雇用一些估算专家，这些专家会使用各式各样的自动化估算工具，并做大量的成本估算。一般情况下，项目办公室会配备一些常见的估算工具，如 COCOMO、KnowledgePlan、Price-S、SEER、SoftCost 以及 SLIM 等，并且还使用这些工具寻找收敛的结果。

正如前面所提到的，除非有类似项目的历史数据可以支持项目的估算，不然的话，即使是再精确的估算，也可能会遭到拒绝。事实上，即使历史数据比凭空的估算更可信，但是项目依旧可能会被拒绝，并被无理的要求所取代。

对于少于 1000 个功能点的小型项目来说，编码仍然是占主导地位的活动。对于这些小型应用来说，自动估算和人工估算的准确率大致相当，不过自动估算会更快一些，并且更容易调整。

2.17 软件项目的风险分析

请不要怀疑，在人类历史上，开发拥有 10 000 个功能点的大型软件项目是最危险的商业活动。大型软件项目的主要风险包括：

- 由于成本超支，并且进度延期，项目最终被彻底取消。
- 由于经济不景气，导致公司缩编或破产，项目最终被彻底终止。
- 与初始估算相比，成本超支了 50%。
- 与初始估算相比，进度延期 12 个月。
- 质量控制是如此不到家，以致软件不能正常有效地工作。

- 每个月的需求变更超过了 2%。
- 管理人员或客户破坏项目进度。
- 客户没法有效地审查需求和规划。
- 安全缺陷和漏洞。
- 性能或速度太慢。
- 项目在开发过程中流失关键的人员。
- 在传统的应用中存在易错模块。
- 专利侵权或窃取知识产权。
- 外部风险（火灾、地震、飓风等）。
- 出售或收购具有相似软件的业务部门。

在违反合同的诉讼中，我们对证词和法庭文件进行了分析，结果发现最失败的项目甚至没有进行正规的风险分析。此外，质量控制和变更管理也是相当地匮乏。更糟糕的是，项目跟踪不是很到家，主要的问题是，当出现问题时，问题被掩盖了。另一个风险是，准确的规划和预算通常都被拒绝，并且被不切实际的规划和预算所代替，这些不切实际的规划和预算通常都是根据业务需求来制定的，而非团队的实力。

最先进的软件风险管理技术是改进。传统情况下，训练有素的风险专家所给出的风险分析能够提供最好的防御。然而，在 2008 年，风险估算工具和软件风险模型在数量和复杂性上都有所增加。计算机辅助公司新式的数据分析工具和笔者的软件风险大师原型（Software Risk Master Prototype）都是预测工具的范例，它们可以量化各种风险的概率和影响。

截至 2009 年，软件风险管理的最佳实践包括：

- 早期的风险评估甚至要赶在全部需求出来之前。
- 潜在缺陷和缺陷去除效率水平的早期预测。
- 将项目的风险模式和类似的项目做比较。
- 从 ISBSG 的数据库获得基准。
- 提前审查合同和质量标准。
- 提前分析变更控制方法。
- 由于经济不景气，需要尽早分析应用的价值。

正规风险管理的重要性随着应用规模的增加而增加。对于 1000 个功能点以下的應用，风险管理通常是可选的；对于超过 10 000 个功能点的应用，风险评估是强制性的；而对于超过 10 万个功能点的应用，未能履行审慎的风险评估则就是专业过失了。

通过对违反合同的诉讼进行反复观察，我们发现那些软件项目几乎从来没有执行过有效的风险评估，因此大家最终不得不在法庭上相见了。相反，虚假的乐观主义和不切实际的进度和成本估算使得项目从第一天起，就朝着一个坏的方向发展。

不幸的是，最严重的风险涉及许多可变的因素。因此，组合的复杂性增加了全面风险分析的难度。当问题有两个以上的可变因素时，光凭人脑是很难处理的。如果可变因素的数量巨大，如超过 10 个，即使是自动化的风险模型可能也会步履维艰。正如经济风险模型在 2008 年失败预测金融危机一样，风险分析并不是一个成熟的领域，并且可能会遗漏严重

的风险。另外，风险分析也会有误报，它们会报告一些实际上并不存在的危险因素，尽管这种情况是很罕见的。

2.18 软件项目的价值分析

2009 年的时候，软件的价值分析并不是很复杂。在开发之前，我们甚至无法量化软件应用的价值，并且即使获得了量化结果，大家对该结果也满腹疑云。

软件应用有经济价值和无形价值两个方面。经济价值可细分为成本降低和收益增加。无形价值的特点是很难描述的，但它涉及的主题却很广泛，如客户满意度、员工的士气，甚至还包括更重要的主题，比如改善人的生活和安全、加强国防等。

一些主题应该包含在价值分析的研究中，比如：

有形的经济价值：

- ☐ 降低新应用的成本。
- ☐ 新应用的直接收益。
- ☐ 由于其他因素，如硬件的销售，使得新应用获得了间接收益。
- ☐ 配套应用（Companion Application）的收益增加。
- ☐ 新应用的国内市场份额增加。
- ☐ 新应用的国际市场份额增加。
- ☐ 新应用的竞争性市场份额下降。
- ☐ 新功能使得用户数量增加。
- ☐ 提升用户性能。
- ☐ 减少用户错误。

无形的价值：

- ☐ 如果你的竞争对手开发了新应用，那么这将成为你潜在的危害。
- ☐ 如果你的竞争对手开发了类似的应用，那么这将成为你潜在危害。
- ☐ 如果你的应用是第一个进入市场的，那么会存在潜在收益。
- ☐ 和已发布的应用保持协同。
- ☐ 国家安全利益。
- ☐ 人类健康或安全利益。
- ☐ 企业信誉带来的价值。
- ☐ 员工士气带来的价值。
- ☐ 客户满意度带来的价值。

笔者提议构建一个价值点指标，该指标在结构上类似于功能点指标，可以整合经济价值和无形价值两方面的主题，并且可以用来计算投资回报率。

在一般情况下，经济价值点等于 1000 美元。无形价值点必须映射到近似等价的规模，如每个客户价值 10 点，客户增加或减少与价值点对应。显然，与拯救人类生命或国防相关的价值需要按对数计算，因为这些价值是无法估量的。

从经济学研究来说,价值点能够和功能点成本进行比较,就像投资和总拥有成本(TCO)的回报率一样。

2.19 取消或拯救陷入困境的项目

目前,虽然有很多大型软件项目都出现了延期或成本超支,但是令人惊奇的是,有关这一主题的文献却非常稀少。当下,有一些有趣的技术性论文,但却没有较为全面的书籍。当然,也有许多书籍讨论软件灾难以及彻底的失败,但它们几乎没有试图拯救陷入困境的项目,或者对最佳实践做一些讨论。

不幸的是,只有一小部分问题项目是可以得救的,并且凤凰涅槃,成为成功的项目。出现这种情况的原因有两个:第一,有问题的项目通常进度跟踪很糟,以至于想拯救项目时,却发现为时已晚,并且问题还在更高层的管理者或客户面前呈现;第二,问题项目遭遇延期和成本超支,使得项目逐渐失去了价值。虽然这些项目在第一次启动的时候会有正收益,但是当第二次或第三次的时候,就会出现成本超支,并且收益也有可能降低,以至于完成该应用也不再具有成本效益了。举个例子澄清一下有关情况。

示例显示了一个原始的估算,并且高级管理人员在知道了原始的估算已经失效的情况下,会接着构建3个后续的估算。对于大型制造企业来说,我们所讨论的应用是一个订单录入系统。它的初步规划规模为1万个功能点。

应用的原始成本约为2000万美元,而原始的价值约为5000万美元。然而,该估算值在应用投入生产36个月后才能生效。每延期一个月都会降低该应用的价值。

估算 1: 2009 年 1 月		估算 3: 2011 年 6 月	
原始的规模(功能点)	10 000 个	预估的规模(功能点)	15 000 个
原始的预算(美元)	20 000 000 美元	预估的预算(美元)	30 000 000 美元
原始的进度(月)	36 个月	预估的进度(月)	48 个月
原始的价值(美元)	50 000 000 美元	预估的价值(美元)	40 000 000 美元
原始的投资回报率(美元)	2.50 美元	预估的投资回报率(美元)	1.33 美元
估算 2: 2010 年 6 月		恢复的可能性很小	
预估的规模(功能点)	12 000 个	估算 4: 2012 年 6 月	
预估的预算(美元)	25 000 000 美元	预估的规模(功能点)	17 000 个
预估的进度(月)	42 个月	预估的预算(美元)	25 000 000 美元
预估的价值(美元)	45 000 000 美元	预估的进度(月)	54 个月
预估的投资回报率(美元)	1.80 美元	预估的价值(美元)	35 000 000 美元
有恢复的可能性		预估的投资回报率(美元)	1.00 美元
		没有恢复的可能性	

从上表中可以看出,需求蔓延的稳定增长引发了开发成本的稳步上升,同时也使得开发时间稳步增加。由于初始的估算时间是36个月,因此在延期之后,该项目的价值被削弱了,并且该项目已不再可行。到了第四个估算的时候,项目恢复的可能性已经为零了,并

且终止项目是唯一的选择。

当然,真正的最佳实践是,在应用程序启动之前,一般会通过执行详细的风险分析及规模研究来避免上述情况的发生。一旦该项目启动,那么协调的最佳实践将包括:

- 仔细精准的状态跟踪。
- 需求出现变更时,需要重新估算项目的进度和成本。
- 每隔一段时间,对项目的价值进行重新估算。
- 考虑无形资产的内部收益率以及经济价值。
- 使用内部的协调专家(如果有的话)。
- 招聘外部的协调顾问。
- 如果应用是根据合同来的,那么坚决抵制诉讼。

当应用开始产生负面价值的时候,试图把它逆转过来简直就是缘木求鱼。那么取消项目的最佳实践包括:

- 挖掘应用中有用的算法和业务规则。
- 提取潜在的可重用代码段。
- 拥有一个正规的检验报告,并知道问题出在哪里。
- 如果应用签订了合同,那么为诉讼准备一些必要的数据。

不幸的是,取消项目是很常见的,但通常不会产生太多有用的数据,以避免类似的问题再次发生。应该将检验报告看作被取消项目的最佳实践。

研究被取消项目的一个困难是,没有人愿意花资金来度量应用程序功能点的规模。然而,高速廉价的功能点度量方法的问世意味着,每个功能点的成本从 6.00 美元下降到了 0.01 美元。假如功能点的成本为 0.01 美元,那么即使是 10 万个功能点的灾难也可以很容易量化。知道了被取消项目的规模之后,我们就可以为软件经济学提供新的见解,并且为项目的分析提供帮助。

2.20 软件项目的组织结构

关于软件项目的组织结构和软件的专业化,相关的讨论多于实践。许多“小团队”理念的拥护者都认为,6 人以下的小团队所开发出来的软件应用在质量和生产率方面更有优势。然而,这样的小团队并不能开发真正的大型应用。

随着软件项目的规模不断增加,企业所雇用的各种专家的数量也在快速上涨。随着企业的人员不断增加,企业的组织结构也变得复杂起来,同时沟通的渠道也呈几何级数增加。这些较大的团队最终变得如此纷繁多样,以至于某些形式的项目办公室被要求跟踪项目的进度、问题、费用以及其他议题。

笔者和其他软件职业组织的同事们(来自大型企业和政府机构)对软件工程行业做了一项调查,发现软件行业共有 75 种不同的专业。由于软件工程并不是有执照的正规专业,所以这些专家的称呼并不能在人事记录中明确地看到。因此为了确定真正的职业,与当地的管理人员一起现场参观和讨论是必要的。

这种情况是比较复杂的,因为有些公司根本无法确定专家的职别或工作形式,所以使用一个通用的标题,如“技术人员”,以包含许多不同的职业。

此外,这样做还增加了探索软件专业化路径的困难,比如我们并不能将开发嵌入式软件的人称作软件工程师,因为他们是电气工程师、自动化工程师、电信工程师或其他一些类型的工程师。在许多情况下,这些工程师都反感别人称自己为“软件工程师”,因为软件工程的专业地位有些低,并且软件工程并不为大家所认可。

让我们分别考虑1000个功能点、1万个功能点以及10万个功能点的应用,它们的人事安排在数量和种类有何差异。对于1000个功能点的小型项目,通才多如牛毛,而专家则寥寥无几。但是,随着应用程序的规模逐渐变大,当功能点达到10 000甚至是100 000的时候,专家就变得越来越重要了,并且在数量上也有很大的提升。表2-3说明了三种不同规模应用的典型人员编制模式。

从表2-3可以很容易地看出,职业的多样性随着应用规模的增加而迅速上升。对于小型应用程序,通才占据了主导地位;但对于大型系统来说,各类专家数量可能达到了团队的三分之一。

表2-3也说明了一些方法的特性,如敏捷开发为什么适合小项目,但对于大型项目却有失妥当。随着项目规模逐渐变大,要想在一个灵活、有凝聚力的团队组织内容纳各种类型的专家已是困难重重。

例如,大型软件项目受益于专门的机构,如项目办公室、正规的软件质量保证(SQA)组织、正规的测试团队、度量团队、变更管理委员会以及其他一些机构。使大型项目受益的专门职业包括体系架构、安全、数据库管理、配置管理、测试以及功能点分析等。

将大型软件项目中不同职业的人员组合成一个有凝聚力、有合作精神的团队并不是很容易。跨部门和跨专业会使团队之间的沟通呈几何数增长。因此,对于拥有1万个功能点以上的大型软件项目来说,最佳实践就是设立一个项目办公室,该办公室的主要职责是协调大型系统的各种技能和活动,是大型系统开发的必要组成部分。简单的小型自组织团队使用敏捷方法只能开发2500个功能点以下的應用。

另一个问题是,需要检查控制跨度,或者向经理报告雇员的数量。由于企业政策的原

表 2-3 软件项目的人员编制模式

工作组	1 000 个 功能点	10 000 个 功能点	100 000 个 功能点
架构		1	5
配置控制		2	8
数据库管理		2	10
估算专家		1	3
功能点计数专家		2	5
度量专家		1	5
规划专家		1	3
项目库管理人员		2	6
项目经理	1	6	75
质量保证		2	12
Srum Master		3	8
安全专家		1	5
软件工程师	5	50	600
技术作家	1	3	12
测试师		5	125
网页设计师		1	5
员工总数	7	83	887
每名员工的平均功能点数	142.86	120.48	112.74

因，在美国，每个经理所管辖的软件雇员的数量大约为 8 个；然而，实际观察到的数量却是 3 到 20 个。

笔者在 IBM 所进行的研究表明，每位经理管理 8 名员工。并且使得更多的人从事管理，这样所得到的效果比只让真正有资格的人来从事管理好得多。因此，规划、估算以及其他管理职能有时也表现欠佳。笔者的研究结论是，将平均控制跨度从 8 个人变成 11 个人，可以使得那些边缘的管理人员重新踏上技术岗位。部门数量的减少也将有助于部门之间的沟通，同时也可以使管理人员能有更多的时间与自己的团队相处，而不是花费过多时间与其他经理讨论。

更糟糕的是，管理人员和技术工人之间的人格冲突有时可能会导致一些出色的技术专家主动离职。事实上，通过对大公司软件工程师的离职面谈进行研究，我们发现了两个令人苦恼的问题：（1）离职数量最大的是最优秀的人才；（2）自愿离职最常见的原因是：“我受不了糟糕的管理。”

在这本书后续的章节里，我们将详细讨论小型团队、大型团队的优缺点，并且也会讨论一些专题，如结对编程等。

2.21 培训软件项目经理

当拥有 10 000 个功能点的大型项目出现严重超期或成本超支时，我们可以肯定的是项目管理一定存在问题。相反，当项目有较高的生产率和质量水平时，出色的项目管理自然是功不可没。大型软件项目中最先进的项目管理知识包括：

1. 估算项目的规模，如功能点。
2. 正规的估算工具和技术。
3. 项目规划工具和技术。
4. 基准技术和行业基准来源。
5. 风险分析方法。
6. 安全问题和安全漏洞。
7. 价值分析方法。
8. 项目度量技术。
9. 里程碑和成本跟踪技术。
10. 变更管理技术。
11. 所有形式的软件质量控制。
12. 人才管理技术。
13. 应用程序（正在开发中）的领域知识。

在全球所有的软件产业中，项目管理是一个薄弱环节，并且可能是最薄弱的环节了。例如，对于失败的软件项目，它们都很少使用功能点来估算项目的规模，并且也不使用正规的估算工具。虽然它们可能会使用一些项目规划工具，但是项目往往会延期或者超支。这说明了一个问题，那就是项目缺乏计划，并且还省略了关键的假设，如正常的需求变更、

员工流失以及由于测试期间所发现的高缺陷率而导致的延期。

外包工程的项目管理比内部开发的项目管理稍显复杂。重要的是要了解客户管理与供应商管理之间的差异。

积极主动地管理项目工作是供应商项目经理的职责所在。他们的工作就是创建项目规划和进度表、制订成本估算、制订跟踪成本、生成里程碑报告，同时还得将项目中的潜在问题告知客户主管（Client Director）以及高级客户经理。

客户主管以及高级客户经理主要职责是以为项目提供各种便利、资金为中心，并且审批或拒绝供应商项目经理的规划和估算。

便利是指，客户主管要为供应商的业务和技术人员提供一定的支持，以便收集需求。客户主管也可以给供应商提供技术文档、办公场所以及工具等，甚至还可以牺牲自己的时间。

资金是指，在项目被公司高管审批之后，客户主管要为项目提供资金支持。

审批是指，客户主管考虑供应商的提案、规划以及估算，然后决定到底是接受还是拒绝，或者要求他们修改后重新提交。

项目失败的主要问题似乎都集中在审批上。不幸的是，客户可能会被供应商乐观的进度估算和承诺所迷惑，以至于轻易批准了供应商们的要求。这样做往往会导致累计超支，当然，出现这种情况的原因还是值得我们深入讨论一下的。

一旦项目启动，除非项目完成，否则该项目将没有任何价值。例如，一个项目的预算是一百万美元，假如出现成本超支，需要额外的10万美元才能完成，那么客户将陷入进退两难的境地。要么取消项目，失去了一百万美元的应计成本；要么给供应商提供额外的10万美元，使项目顺利完工，这样才能获得收益。

如果这种情况反复出现，客户的选择将变得更加困难。假如一个项目的应计成本（Accrued Cost）为500万美元，那么不管是选择取消，还是选择继续投入10%的额外费用，供应商的代价都是很昂贵的。这个问题的关键是项目的失败会造成难以估量的损失。预算每修订一次，供应商都会说，该项目的完工指日可待，只需要少量的时间和些许额外的资金即可。但是这种情况还可能再次出现。

对于所有重大的投资，企业都有一定的筹资标准。项目只有获得了一定的收益后才能得到一定的资助。该项目要么可以增加收入，要么可以降低成本或者具有一定的竞争优势。在美国，软件项目典型的投资回报率（ROI）是3:1。也就是说，在项目上每投入1.00美元，可以获得3.00美元的收益。

在开发的过程中，需要对应计成本进行监控。如果成本开始超出预算，那么项目的投资回报率会有所降低。不幸的是，对失败的项目，供应商所使用的不准确的估算方法和成本控制方法是罪魁祸首，它们影响了高级客户经理预测投资回报率的准确性。

当然，问题的根源是估算方法不切实际，以至于他们总是过于乐观。不幸的是，在实际情况浮出水面之前，这种情况会反复出现好几次。

在供应商向客户提出修订估算和进度时，供应商可能并没给客户披露内部所存在的问题和风险。有时，当出现诉讼时，这种问题才会浮出水面，供应商所有的记录都会被披露，并且供应商的工作人员会被免职。

一个项目的首要任务是，改进对项目经理的培训规划，该任务包括估算、状态报告、成本跟踪以及问题报告等。

2.22 培训软件技术人员

软件开发和维护领域的特点是，员工在核心活动上通常具有相当强烈的工作热情以及合理的竞争力，这些核心活动包括：详细设计、编程以及单元测试等。许多软件人员在基本的编程任务上都会投入相当长的时间。但是，要想使 10 000 个功能点以上的应用取得成功，以下所罗列的一些额外技能则是必需的：

1. 应用程序所在的领域
2. 数据库软件包、形式、工具以及产品
3. 外包公司的技能
4. 联合应用设计 (JAD) 的原则
5. 正式设计审查
6. 复杂性分析
7. 使用的编程语言
8. 安全问题和安全漏洞 (薄弱环节)
9. 性能问题和瓶颈 (薄弱环节)
10. 正式代码审查
11. 静态分析方法
12. 复杂性分析方法
13. 变更控制的方法和工具
14. 性能的度量和优化技术
15. 测试方法和工具

当软件出现技术问题时，问题的原因往往是技术人员缺乏软件应用领域相关的专业知识，或者是相关的技术主题，如性能优化等，而不是缺乏软件开发方法的基本知识。

也有可能是缺乏关键质量控制活动方面的知识，如审查、JAD 以及专门的测试方法等。在一般情况下，常见的编程任务并不是什么问题。出现问题的地方往往是需要专业知识的地方，这也引出了我们即将要讨论的下一个专题。

2.23 使用软件专家

在许多人类的活动中，专业化往往是技术成熟的标志。例如，医学、法律以及工程都包含数十种类型的专业。软件像没有成熟的专业一样，复杂而多样，并且其专业化的数量却在持续增加。AT & T 对大型软件生产企业的入口数据进行了统计，分析的结果表明，目前的软件产业中存在 20 ~ 90 种专门的职业。

对于拥有 10 000 个功能点的项目来说，专业化会显得更加重要，这是因为与数量众多的

通才相比,项目拥有数十位有能力的专家会有更高的成功率。

对于拥有 10 000 个功能点的项目来说,最先进的专业化包括以下专家工作组:

1. 配置控制专家
2. 成本估算专家
3. 客户联络专家
4. 客户支持专家
5. 数据库管理专家
6. 数据质量专家
7. 决策支持专家
8. 开发专家
9. 领域知识专家
10. 安全专家
11. 性能专家
12. 培训专家
13. 功能点专家(认证的)
14. 图形用户界面(GUI)专家
15. 人为因素专家
16. 集成专家
17. 联合应用设计(JAD)专家
18. Scrum Master(敏捷开发项目)
19. 度量专家
20. 维护专家(发布后的缺陷修复)
21. 维护专家(小规模的功能增强)
22. 外包估算专家
23. 软件包估算专家
24. 性能专家
25. 项目成本估算专家
26. 项目规划专家
27. 质量保证专家
28. 质量度量专家
29. 可重用专家
30. 风险管理专家
31. 标准专家
32. 系统分析专家
33. 系统支持专家
34. 技术写作专家
35. 测试专家

36. 工具专家（开发和维护工作台）

高级项目经理应该知道什么样的专家是必需的，并且会采取积极而充满活力的手段找到他们。专家通常比通才具有优势的领域包括技术写作、测试、质量保证、数据库设计、维护、性能优化等。对于某些任务，如功能点分析，认证考试是从事该工作的必要条件。对于真正的大项目来说，规划和估算专家也是相当有利的。

目前软件开发和软件项目管理过于庞大，并且也过于复杂，以至于通才没法对知道的东西做足够深入的研究。当下，越来越多的软件专家说明了一个问题，那就是软件工程和软件管理的知识体系在不断扩大，同时软件的形势一片大好。

在过去的 30 年里，美国 and 欧洲的一些公司经常将软件的开发、维护以及技术支持（Help-Desk）活动外包给劳动力成本低廉的国家，如印度、中国、俄罗斯等国家。但重要的是，外包供应商使用了同样的内部开发方法，并且还实行了出色的质量控制。

国际软件基准组织（ISBSG）最近对外包实践做了一项有意思的研究，研究发现，与内部开发的项目相比，外包的开发项目倾向于使用更多的工具以及较为复杂的规划和评估方法。这与笔者自己的观察是一致的。

2.24 软件工程师、专家以及管理人员的认证

在 2008 年年底到 2009 年年初，这本书在编写的时候，软件工程自身以及许多与其相关的专业都尚未明确定义。在整个软件领域里的 90 多个专业中，有认证的可能只有十几个专题。对于这些专题，认证都是自愿的，并且不具有法律地位。

使整个行业受益的做法是建立一个联合认证委员会，该委员会的成员来自大型专业协会，如 ASQ、IEEE（电气和电子工程师协会）、IIBA（国际商业分析研究所）、IFPUG（国际功能点用户组织）、PMI（项目管理协会）以及 SEI（软件工程研究所）等。联合认证委员会将确定专家的种类以及认证的标准。这些标准可能是考试或认证委员会，就像医学专业一样。

在编写这本书的时候，自愿性认证可能包含如下这些主题：

- ☐ 功能点分析（IFPUG）
- ☐ 功能点分析（COSMIC）
- ☐ 功能点分析（芬兰）
- ☐ 功能点分析（荷兰）
- ☐ 微软认证（各种主题）
- ☐ 六西格玛绿带
- ☐ 六西格玛黑带
- ☐ 认证的软件项目经理（CSPM）
- ☐ 认证的软件质量工程师（CSQE）
- ☐ 认证的软件测试经理（CSTM）
- ☐ 认证的软件测试专业（CSTP）
- ☐ 认证的软件测试师（CSTE）

□ 认证的范围经理 (CSM)

这些形式多样的认证由不同的组织提供，他们一般不承认其他组织的认证。对于所有形式的认证，既没有一个统一的认证，也没有一个标准考试。

这样导致的结果是大家缺乏注册认证相关的统计数据，并且也没有各种认证专家和管理人员实际比例的可靠信息。对于技能，如功能点分析，大概有 80% 的顾问和员工通过了认证。对于六西格玛，大概有 80% 以上的顾问和员工通过了认证。然而，对于测试、项目管理以及质量保证，如果认证的比例达到了 20% 以上，那么这一定令人很诧异。

有趣的是，无论是“软件工程”，还是“软件维护”，都没有一个统一的专业认证。

如果软件行业成立统一的认证委员会，那么率先获得认证的技术主题将包括：

- 软件架构
- 软件开发工程
- 软件维护工程
- 软件质量保证
- 软件安全保证
- 软件性能分析
- 软件测试
- 软件项目管理
- 软件范围管理

专业技能也需要通过认证，包括但不限于以下这些主题：

- 软件六西格玛
- 质量功能展开 (QFD)
- 功能点分析 (各种形式)
- 软件质量度量
- 软件生产力度量
- 软件经济分析
- 软件审查
- SEI 评估
- 供应商的认证 (如微软、甲骨文、SAP 等)

正如书中所说，最根本的是，软件认证是自愿的、分散的，并且无论对从业者还是该行业来说，其价值也是未知的。观察表明，在功能点分析和认证计数等技能的准确度上，通过认证的从业者比自学成才的从业者更有优势。然而，认证在软件质量和测试上的优势也需要进行更多的研究。

当下真正需要的是对各种认证进行协调，并设立联合认证委员会，该委员会的职责是考虑各种形式的软件专业化。就像医学和法律界一样，软件工程领域可以采取类似的政策和做法，并且考虑如何创建和管理这些专业。

对于许多形式的认证，目前并没有量化的数据能够说明认证可以提高工作效率。然而，对于个别形式的认证，我们却有足够可用的数据说明认证对软件行业确实有很大的改进：

1. 通过认证的测试人员比未通过认证的人员在缺陷去除效率上高 5% 左右。
2. 在对应用进行功能点计数试验的时候，通过认证的从业者在功能点计数分析上所得到的误差不会超过 5%，而未经认证的从业者所得到的误差则超过了 50%。
3. 假如认证的六西格玛黑带是软件开发团队的一部分，那么应用程序往往会有较低的潜在缺陷，每个功能点大约有 1 个缺陷，并且有较高的缺陷去除效率，大约为 97% 左右（美国平均水平是：每个功能点大约有 5 个缺陷，而缺陷去除效率大约为 85% 左右）。

不幸的是，在写这本书的时候，其他形式认证的量化结果都是模棱两可的。很显然，那些对他们的工作很在意，并且认真学习，顺利通过笔试的人，往往比那些没有通过笔试的人出色，但是这样的结果很难通过量化数据来证实，因为可用的数据屈指可数。

对软件行业有利的方式是，学习美国医学协会的模式，设立一个专门用来识别和认证专家的独立组织，而不是多个甚至是有竞争关系的组织。

2.25 软件项目中的沟通

对于拥有 10 000 个功能点的大型软件应用来说，开发团队的规模大约为 50 到 100 人。此外，大型软件应用经常需要数十甚至数百个用户的参与，并且他们中的大多数将以特有的方式参与应用的构建。

不管构建任何大型和复杂的产品，除非使用了先进的沟通渠道，不然几十个员工和几十个用户根本无法共享信息。当项目涉及多个承包商时，需要的流量可能还会更大一些。

2009 年，一种新型的虚拟环境开始进入到业务领域；在这个虚拟现实世界中，参与者通过虚拟化身进行交互。虽然在 2009 年这种用途还仅仅是限于试验，但是这种用途却正在迅速成为主流。随着飞行成本的日益飙升以及经济状况江河日下，像虚拟通信这样的方法很可能会迅速成为人们的首选。十年之内，这种方法很可能会超过现场会议和实时会议。

项目间的沟通也越来越多的另一个地方是“维基网站”，这是一个协作网络，它可以让同事们相互交流思想、文档以及工作产品等。

对于拥有 10 000 个功能点的项目来说，最先进的沟通技术包括：

- 每个月向企业高管报告一次项目管理的状态。
- 供应商项目经理每周向客户报告一次进度。
- 客户与总承包人之间每日沟通一次。
- 主承包商与分包商之间每日沟通一次。
- 开发人员和开发管理者之间每日沟通一次。
- 使用虚拟现实技术进行通信以跨越地域的限制。
- 使用“维基”网站进行通信以跨越地域的限制。
- 开发团队每天进行一次 Scrum 会议，以讨论相关的问题。
- 在所有的参与者之间全面支持 E-mail。
- 在所有参与者之间全面支持语音。
- 在偏远地区，通过视频会议进行通信。

□ 在开发人员中，自动分配文档和源代码。

□ 给开发人员自动分配变更请求。

□ 给开发人员自动分配缺陷报告。

□ 对于有重大影响的问题，采取紧急沟通。

对于失败的项目来说，要么没有完全采用以上这些交流的渠道，要么团队之间存在间隙，以至于阻碍了团队间的交流和项目的进度。例如，在突出问题上，跨供应商的交流显得捉襟见肘。此外，提交给高管们的状态报告可能也会掩盖问题的本质，以图隐瞒它们，而不是向它们说明工程延期以及工程超支的原因。

良好沟通的根本目的是什么？ITT公司的前董事长 Harold Geneen 做了一个很好的总结，那就是“没有惊喜（No Surprise）”。

通过查看违反合同的诉讼，笔者在证词和法庭文件中发现了一个令人震惊的情况，那就是许多项目都没有状态跟踪和问题报告。通常情况下，被取消或有重大超支的项目甚至都没有在意一些问题，直到问题变得严重起来，想要纠正它们的时候，才发现为时已晚。与此相反，成功的项目几乎没有严重的问题，除此之外，成功的项目还有更加有效的跟踪和更加有效的风险消减计划。当问题首次出现时，成功的项目往往会立即启动风险恢复活动（Risk-Recovery Activity）。

2.26 软件的可重用性

至少有 15 种软件构件本身具有可重用性。遗憾的是，许多软件复用相关的文献只是集中在源代码重用上，而其他方面的重用，如设计重用，则很少有文献做过专门的讨论。

开发 10 000 个功能点的大型项目将需要大量可重用的材料。以下是软件项目中 15 个潜在的可重用构件：

1. 架构
2. 需求
3. 源代码（零缺陷）
4. 设计
5. 帮助文档
6. 数据
7. 培训材料
8. 成本估算
9. 界面截图（Screen）
10. 项目计划
11. 测试计划
12. 测试用例
13. 测试脚本
14. 用户文档

15. 人机交互

不仅构件可以重用,技术和商业上的成功也可以重用,需要记录的信息包括:

- ☐ 所有的客户或用户的信息(以备召回之需)
- ☐ 所有可重用构件中的错误或缺陷
- ☐ 所有版本的可重用构件
- ☐ 可重用材料的认证结果
- ☐ 所有更新或变更

此外,不能重用错误很多的材料。因此,对于成功的重用,完善的质量控制措施是必需的,这些措施有(但不限于以下这些):

- ☐ 审查可重用的文本文档。
- ☐ 审查可重用的代码段。
- ☐ 对可重用的代码段执行静态分析。
- ☐ 测试可重用代码段。
- ☐ 可重用材料的公开认证证书。

成功的软件重用并不是简单地复制一个代码段,然后将其插入到一个新的应用程序中。

外包供应商有一个共同的优势,那就是供应商往往具有成熟的重用技术,并且可以提供许多可重用构件。因而,假如供应商非常专业的话,那么重用将经常出现。例如,一个专门从事保险应用的外包供应商,他可能已经与十几个财产和伤亡保险公司合作过,并且积累了大量的可重用材料,因此在构建任何保险应用的时候,他至少可以使用 50% 以上的可重用组件。

软件复用是降低成本和进度,并且提高产品质量的关键因素。然而,重用也是一把双刃剑。如果可重用材料的质量水平是无可挑剔的,那么可重用性对任何已知的软件技术都会有最高的投资回报率。但是,如果重用材料中有很多的错误,那么投资回报率可能会变成负值。事实上,高品质重用和低品质重用之间的差异还是蛮大的,通过观察发现,任何已知技术的投资回报率都为 $\pm 300\%$ 。

软件的可重用往往被视为是灵丹妙药,既可以弥补软件开发的缓慢进度,也可以降低软件的高昂成本。这在理论上可能是行得通的,但是除非可重用材料的质量接近零缺陷,否则可重用并没有任何实用价值。

在过去的数年里,出现了一种新型的可重用,即面向服务架构(SOA)。SOA 方法处理重用的途径是,将相对独立的功能或“服务”组装成一个内聚的应用。函数本身也可以在单机模式下运行,并且不需要做任何修改。SOA 是一个有趣的概念,并且也给软件行业带来了新的生机,但是截至 2009 年,SOA 的概念基本上都是理论上的,而实际的应用却很少。目前,SOA 在成本、质量以及有效性方面的经验数据都几乎为零。

到目前为止,软件的可重用并没有达到如大家所期望的那样。无论是面向对象类库,还是其他形式的重用,如商业企业资源规划(ERP)套件,都已经有了成功的实践。

要想将可重用提升到对经济非常有利的地位,那么可重用材料就需要有更出色的质量以及更卓越的安全控制才行。当下,可重用的技术似乎已准备妥当,因此,或许在未来的

几年里,可重用终将进入大家的视线。

要想将软件放置良好的经济基础之上,软件的范式需要做一些变化,即将软件开发从使用定制代码转换成使用标准的可重用的组件。截至2009年,只有很少的应用使用标准可重用组件进行构建。其中一部分原因是,许多软件组件的质量控制做得不够好。另一部分原因是,缺乏常规的应用类型的标准架构,并且还缺乏连接组件的标准接口。在当前典型的应用中,高品质可重用材料的使用率甚至低于25%。因此我们要做的就是脚踏实地,逐步规划,不断提升高质量可重用材料的数量,使得常规应用中可重用材料的比例达到85%以上,甚至是95%以上。

2.27 可重用材料的认证

编码、规格和其他资料的重用也是有利有弊的。如果该软件没有任何技术漏洞且已经成熟使用,那么它能够带来的利润就可以比任何已知的科技都丰厚。但是,如果该软件有很多漏洞并且还未成熟使用,那么继续使用只会应用中增加更多的漏洞。在这种情况下,软件的可重用比任何已知科技带来的负面影响都大。

由于可重用材料的数量巨大,并且部分可重用材料的可靠性还未知,因此到目前为止,在软件中使用可重用材料还不是一个成熟的做法。此外,有的可重用材料可能还存在安全上的问题,如软件源代码,有的甚至还被黑客植入了木马。黑客们可能使用这些漏洞来敲诈公司,以获得不法利益。

这就引出了一个非常重要的问题:怎样才能软件中放心地使用低廉、高效的可用材料,并为企业带来利润?

首先,得确保有一个大家信服的机构,该机构可以出具证明或者多重证明来保证可重用软件基本上无漏洞、无病毒、无间谍软件,并还没有按键记录器(keystroke logger)。该机构最好是一个非营利组织,并且由企业出资,就像美国保险商实验室和美国消费者报告之类的组织。

但是仅仅有源代码的有效证明是不够的。因此要想成功使用可重用材料,以下一些专题也是需要考虑的:

- 可重用材料的正规分类及用途
- 连接可重用材料的标准接口的定义
- 所有可重用材料的用户信息以及使用说明
- 所有可重用材料相关的测试用例以及测试脚本
- 所有可重用材料的错误报告
- 可重用材料的来源确认
- 所有可重用材料的变更记录
- 所有可重用材料的种类
- 所有可重用材料的分布记录(以备召回之需)
- 对非免费的可重用材料实行收费
- 确保可重用材料的版权和知识产权不受侵犯

也就是说,如果可重用的部分是软件的关键部分,那么就需要将不稳定的非正式版本升级成为稳定的正式版本。只有这样,可重用才能更有价值;不然的话,今后将会为软件留下巨大的安全隐患。若软件行业中设有一个专门的非营利中心,并且该中心为可重用材料提供来源,那么整个行业都将从中受益。

表 2-4 显示了高品质可重用材料的开发价值,这些高品质材料都是经过认证的,并且接近零缺陷。表中所呈现的可重用材料不仅有编码,还有架构、需求、设计以及测试材料和文件。表 2-4 中的例子是一个拥有 10 000 个功能点的大型系统,一般情况下,这种规模的项目约有一半会以失败告终,并且质量也会严重不达标。正如我们在表中所看到的,当可重用材料的百分比在上涨时,软件的生产率和质量都有显著提升,同时开发进度也有所提升。

表 2-4 高品质可重用材料的开发价值

应用规模 (功能点) = 10 000							
重用比例	员工数	工作量 (月)	生产率 (功能点 / 月)	进度 (月)	潜在缺陷	缺陷去除效率	交付缺陷
0.00%	67	2 654	3.77	39.81	63 096	80.00%	12 619
10.00%	60	2 290	4.37	38.17	55 602	83.00%	9 452
20.00%	53	1 942	5.15	36.41	48 273	87.00%	6 276
30.00%	47	1 611	6.21	34.52	41 126	90.00%	4 113
40.00%	40	1 298	7.7	32.45	34 181	93.00%	2 393
50.00%	33	1 006	9.94	30.17	27 464	95.00%	1 373
60.00%	27	736	13.59	27.59	21 012	97.00%	630
70.00%	20	492	20.33	24.6	14 878	98.00%	298
80.00%	13	279	35.86	20.91	9 146	98.50%	137
90.00%	7	106	94.64	15.85	3 981	99.00%	40
100.00%	4	48	208.33	12	577	99.50%	3

在软件经济中,任何已知的技术都不可能像高品质可重用材料一样取得如此巨大的效益。而可重用便是面向对象开发和面向服务开发的目标所在。只要软件应用的编码是通过定制来完成的,那么产品生产率和质量的提高都将限制在 25% 到 30% 之间。若想获得上百个百分比的增长率,采用高质量的可重用材料是最佳途径。

高质量可重用材料不仅能够使开发受益,而且还能够使维护和改进得到改善。然而,软件维护需要注意一个问题,那就是,在成千上万的产品中使用了可重用材料之后,我们一定得做好记录,为日后的跟踪、升级或者修复漏洞做准备。所以,要获得最大的经济价值,可重用材料的认证以及详尽的使用记录则是必需的。本书中的“维护”是指缺陷修复,而“功能增强”则是指为软件增加新的功能。

表 2-5 说明了可重用材料的维护价值。

软件中可交付缺陷与软件开发人员和维护人员息息相关,因此当认证的可重用材料的使用量上升时,这些人员的需求量也会相应地下降。

可交付缺陷也会受到客户支持的影响,当然还有很多其他的影响因素。除了以上提到的影响可交付缺陷的因素之外,其他影响可交付缺陷的因素主要有用户的数量以及软件的安装量。

表 2-5 高品质可重用材料的维护价值

应用规模 (功能点) = 10 000							
重用比例	员工数	工作量 (月)	生产率 (功能点 / 月)	进度 (月)	潜在缺陷	缺陷去除效率	交付缺陷
0.00%	13	160	62.5	12.00	12 619	80.00%	2 524
10.00%	12	144	69.44	12.00	9452	83.00%	1 607
20.00%	11	128	78.13	12.00	6276	87.00%	816
30.00%	9	112	89.29	12.00	4113	90.00%	411
40.00%	8	96	104.17	12.00	2393	93.00%	167
50.00%	7	80	125	12.00	1373	95.00%	69
60.00%	5	64	156.25	12.00	630	97.00%	19
70.00%	4	48	208.33	12.00	298	98.00%	6
80.00%	3	32	312.5	12.00	137	98.50%	2
90.00%	1	16	625	12.00	40	99.00%	0
100.00%	1	12	833.33	12.00	3	99.50%	0

总的来说,一个客服人员需要应对大约 1000 个客户(当然这也不一定就是最适合的比率,因为假如没有长期的研究,这样的比率调查是没有意义的)。当然如果一名客户支持人员应对 150 名客户的话,客户的等待时间将大大缩短,但与此同时成本却大大提升。一般情况下,客户支持服务都会外包给劳动力比较低廉的国家,因此公司每月在这方面的成本大约会从 10 000 美元下降到 4000 美元。

通常情况下,客户少的小公司会比客户多的大公司拥有更好的客户服务,这是因为小公司的客服人员是不饱和的。

表 2-6 说明了当软件中的可重用数增加时,客户支持的近似价值变化。该表假定安装量是 500 个,并且用户数是 25 000 人。

表 2-6 高品质可重用材料客户支持的价值

应用规模 (功能点) = 10 000 安装量 = 500 用户数量 = 25 000							
重用比例	员工数	工作量 (月)	生产率 (功能点 / 月)	进度 (月)	潜在缺陷	缺陷去除效率	交付缺陷
0.00%	25	300	33.33	12.00	12 619	80.00%	2524
10.00%	23	270	37.04	12.00	9452	83.00%	1607
20.00%	20	243	41.15	12.00	6276	87.00%	816
30.00%	18	219	45.72	12.00	4113	90.00%	411
40.00%	16	197	50.81	12.00	2393	93.00%	167
50.00%	15	177	56.45	12.00	1373	95.00%	69
60.00%	13	159	62.72	12.00	630	97.00%	19
70.00%	12	143	69.69	12.00	298	98.00%	6
80.00%	11	129	77.44	12.00	137	98.50%	2
90.00%	10	116	86.04	12.00	40	99.00%	0
100.00%	9	105	95.6	12.00	3	99.50%	0

由于绝大多数客户支持涉及产品的质量问题的,因此提高产品质量本质上会对客户支持成本产生很大的影响,有时候还可能会提高客户满意度,并且减少客户的服务等待时间。

认证的可重用材料也会使产品的改进受益良多。一般情况下,产品的改进每年会提高 8 个百分点。也就是说,假如一个产品的功能点是 10 000 的话,那么下一年该产品将增加 800 个功能点。这并不是一个恒定不变的数值,并且这种改进量会经常变化,但是 8 个百分点仍然是一个非常有用的近似值。表 2-7 显示了不同百分比的可重用材料给产品改进带来的影响。

表 2-7 高品质可重用材料的改进价值

应用规模 (功能点) = 10 000								
功能增强 (功能点) = 800								
使用年限 = 10								
安装量 = 1 000								
用户数 = 50 000								
重用比例	员工数	工作量 (月)	生产率 (功能点 / 月)	进度 (月)	潜在缺陷	缺陷去除效率	交付缺陷	
0.00%	6	77	130.21	12.00	3046	80.00%	609	
10.00%	5	58	173.61	12.00	2741	83.00%	466	
20.00%	4	51	195.31	12.00	2437	87.00%	317	
30.00%	4	45	223.21	12.00	2132	90.00%	213	
40.00%	3	38	260.42	12.00	1828	93.00%	128	
50.00%	3	32	312.5	12.00	1523	95.00%	76	
60.00%	2	26	390.63	12.00	1218	97.00%	37	
70.00%	2	19	520.83	12.00	914	98.00%	18	
80.00%	1	13	781.25	12.00	609	98.50%	9	
90.00%	1	6	1562.5	12.00	305	99.00%	3	
100.00%	1	4	2500	12.00	2	99.50%	0	

尽管大部分总拥有成本受制于缺陷移除成本和维护成本,但是也还有其他一些影响总拥有成本因素。表 2-8 显示了开发以及后续的 10 年使用期中的两类假设结果,分别是无可重用材料和有 80% 可重用材料的情况。很显然,在表 2-8 中将总成本过于简单化了,但其目的在于显示认证的高品质可重用材料能够带来的显著经济价值。

软件中全部使用可重用材料是不太可能的。然而,实验证明,如果为软件提供充足的认证组件,几乎所有的软件的可重用率都可以达到 85% 甚至是 90%。许多年前在 IBM 公司做了一项有关财务软件可重用性的研究,研究表明,85% 的程序编码都是通用的,这其中包括将财务数据输入电脑时所需的符号编辑。只有大约 15% 的程序代码涉及财务处理本身。

可重用材料不仅包含编码,还包含体系构架、需求、设计、测试材料、使用指南以及其他一些必要的组成部分。这些部件都需要经过配置控制以及零缺陷认证才能发挥其最佳使用价值。近些年来,软件可重用技术一直都有很好的市场,但是却从未真正发挥过它真正的潜力,这主要是因为质量监控的落后。如果能够构建以服务为导向的体系构架,那么认证的可重用材料就可以达到最佳的质量水平,并且得到充分利用。

表 2-8 高品质可重用材料的总拥有成本 (0% ~ 80% 可重用量)

应用规模 (功能点) = 10 000			
年改进量 (功能点) = 800			
每月成本 = 10 000 美元			
支持成本 = 4000 美元			
使用年限 (部署后) = 10 年			
	0% 重用	80% 重用	差
开发	26 540 478 美元	2 788 372 美元	-23 752 106 美元
改进	7 680 000 美元	1 280 000 美元	-6 400 000 美元
维护	16 000 000 美元	3 200 000 美元	-12 800 000 美元
客户支持	12 000 000 美元	5 165 607 美元	-6 834 393 美元
总成本	62 220 478 美元	12 433 979 美元	-49 786 499 美元
总拥有成本 (每个功能点)	3 456.69 美元	690.78 美元	-2765.92 美元

除了要达到近乎于零缺陷的质量水平外,也应该开发和设计认证组件,使其更加安全可靠,能够防止黑客、僵尸网络以及病毒的入侵,并且还能避免其他安全隐患。实际上,在开发认证的可重用材料时,一个强有力的例证就是使用更好的边界控制以及更加安全的程序语言,比如说 E 语言,这些方法都会使认证的可重用材料的价值发挥到极致。

随着全球经济发展的衰退,所有的公司都在寻求降低成本的方法。由于软件的成本历来都很高并且也难以控制,因此当前不景气的经济可能会使很多企业被迫采用软件重用的方法,以降低软件成本。当然,要使重用软件发挥其最大的价值,那么质量和安全认证都是非常重要的环节。

2.28 编程

尽管在软件开发过程中,最昂贵的不再是编程或编码,但它们仍然是最核心的活动。尽管大家都想用面向对象开发(OO)、应用程序生成器、面向服务的架构(SOA)以及其他一些软件重用的方法来代替手工编码,但是截至 2009 年,许多软件项目在很大程度上还是依赖手工编码。

由于手工编码很容易出错,而编码又严重依赖于人工操作,因此软件开发自然成了所有产品中最昂贵的一个环节了。

还有很多产品都像这样,它们对精细的人工工艺要求很高,比如说建造 12 米高的邮轮、F1 方程式赛车等。建造同等排水量的定制豪华游轮的成本会比普通邮轮的成本高 10 倍左右。尽管重复使用了一些材料,但是建造 F1 赛车所需的成本还是要比在生产线上批量生产的轿车高出近 100 倍。

与其他工程学科不同的是,目前已知的编程语言已经达到 700 种了。不仅如此,有些大型软件应用还同时使用 12 ~ 15 种不同的编程语言。这主要是因为多数编程语言的使用领域过于专一和狭窄。因此,如果一个应用程序拥有很多功能,那么它肯定会使用很多种编程语言。20 世纪 70 ~ 80 年代,最常见的组合是 COBOL 语言和 SQL 语言;到了 20 世

纪 90 年代,就变成了 Visual Basic 语言和 C 语言;而当前,Java Beans 语言和 HTML 语言的组合则是最流行的。

在过去的 30 年里,编程语言以每月一门的速度迅速增加。目前的编程语言表由软件生产力研究所 (Software Productivity Research, SPR) 维护,该表包含 700 多种编程语言,并且还会增加 12 种新的编程语言。更多信息可参考网站: www.spr.com。

截至 2009 年,最佳的编程方法包含以下主题:

- ☐ 编程语言的选择应与程序应用的需求相吻合。
- ☐ 对于过程式编程,使用结构化程序设计方法。
- ☐ 在开始编码前选择认证的可重用代码。
- ☐ 设计过程中应当考虑安全问题,包括使用安全编程语言,如 E 语言。
- ☐ 避免使用错综复杂的代码。
- ☐ 尽量减少代码的循环复杂度和基本复杂度。
- ☐ 在源代码中包含清楚的注释信息。
- ☐ 对 Java 及 C 语言使用自动化静态分析工具。
- ☐ 在编码之前或在编码的同时创建测试用例。
- ☐ 对所有的模块都执行代码审查。
- ☐ 重大变更或程序升级后,对代码再次进行审查。
- ☐ 在对主要功能进行改进时,先对遗留代码进行改造。
- ☐ 从遗留代码中去除易错模块。

美国商务部并未将编程划归为一个单独的职位,而是将其看作一项工艺或专业技术服务。优秀的编程也具有艺术品的某些特质。所以说个人的技能以及细致的专业训练都会对软件的质量和适用性产生很大的影响。

行业内经常使用“结对编程”,该方法一般是指两名编程人员共同编写同一段代码。其中的一人负责编写代码,另外一人负责审查。大量的实例研究表明,这种做法会使编程的质量有很大提升。然而,这种做法却降低了编码的效率。一种较为好的做法是,先由一个人负责写程序,然后再通过静态分析或者同行审核,这样得到的编码质量也会高于平常,并且所花费的成本会小于“结对编程”。

事实证明,人们往往不善于查找自身的错误,总是自以为是地认为自己的代码不存在漏洞和错误。因此同行审查或者由其他专业的人员来检查,通常是一种值得称赞的做法。

如果软件的开发始终以定制为主,那么整个软件的成本以及错误率将会一直居高不下。只有使用高品质的可重用材料才能使软件行业有根本性的改变,这些改变主要体现在软件成本、设计、质量水平以及错误率上。

很明显,如果软件应用程序是由可重用材料组成的,那么每一个可重用组件的成本就会比现在高很多,因为这些额外的成本主要用来实现尖端的安全控制技术、优化组件的性能水平、创建最先进的规则和用户手册以及构建零缺陷组件等。即使可重用材料的成本是目前定制编码成本的 10 倍以上,但是假如可重用材料重复使用 100 次,那么它的成本也只会有关成本的 1/10。

2.29 软件项目管理

在过去的几年里，有大量的公司高层涉嫌内线交易、财务违规以及其他一些违法行为，有的甚至用假债权欺骗股东。因此，美国国会于2002年通过了萨班斯-奥克斯利法案(SOX)，该法案已于2004年正式生效。

《萨班斯-奥克斯利法案》适用于年收入超过七亿五千万美元的公司。此法案要求对公司的高管们实行严格的责任制。因此，在大公司内，管理这个议题越来越受到重视了。

在“管理”这个概念下，高级执行官将不再是公司财政事务或者财务软件的被动观察者了，这些财务软件一般会包括公司的财务数据。执行官必须积极主动地监督管理各种主要的财务问题以及股票交易情况，并且还使用软件管理公司簿记。

除此之外，为了改善公司的财务管理，还必须提供一些补充报告和数据，这样才能确保公司有绝对的公平、公正，并且责任制得以兑现。若公司执行官们不遵守《萨班斯-奥克斯利法案》，那么他们可能会面临着重罪指控，该指控意味着长达20年的监禁以及高达50万美元的罚金。

由于《萨班斯-奥克斯利法案》只适用于年收入大于七亿五千万美元的大型上市企业，小企业和私人公司并不受制于这个法案。然而，由于公司的高管们过去有诸多的不当行为，因此公司现在对他们的诚信度有了更高的要求。所以，管理依然是一个非常重要的议题。

《萨班斯-奥克斯利法案》在执行初期，政府组织了一个25~50人左右的团队，该团队由企业执行官和信息技术专家组成；经过一年多的努力，他们总结出了《萨班斯-奥克斯利法案》的一系列实施框架。目前，许多的财务软件都需要修改，当然这些修改必须在《萨班斯-奥克斯利法案》的框架内。在不久的将来，可能还需要20名专职人员的不断投入，才能使该法案得到很好的实施。由于SOX法律严格的规定以及对违规行为的严重处罚，因此多数企业都需要快速去适应《萨班斯-奥克斯利法案》的规定。司法建议对企业来说也是非常重要的。

由于企业管理和《萨班斯-奥克斯利法案》都是很重要的，因此许多咨询公司都提供管理以及《萨班斯-奥克斯利法案》的咨询业务，当然也有很多软件工具可以帮助企业进行监管和控制。因此，在《萨班斯-奥克斯利法案》生效前，企业管理人员需要了解并熟练应用财务软件。

如果对软件的管理不当，那么美国大型软件公司有可能会遭到起诉或罚款。然而，正确的软件管理也不仅仅限于大型上市公司，政府机构、小型企业以及在其他国家的公司不但不会受到《萨班斯-奥克斯利法案》的影响，并且还会从软件管理的最佳实践中受益。

2.30 软件项目的度量指标

一流企业一般会通过软件测量程序来获得生产率和质量方面的历史数据。对于拥有10 000个功能点的应用来说，项目最先进的软件度量措施包括：

1. 累积成果
2. 累积成本

3. 完成选定的里程碑
4. 开发效率
5. 维护和功能增强效率
6. 需求变更的数量
7. 源代码中的缺陷
8. 缺陷去除效率
9. 挣值（主要用于国防项目）

度量的工作量通常是细粒度的，并且还支持 WBS(工作分解结构)。成本度量是完整的，其中包括开发成本、合约成本以及与采购、租赁相关的成本。有一个模糊的问题是，即使对于一流的公司来说，软件成本中的日常开支和分摊率在不同的软件中也有很大差别，而且还会扭曲公司、行业以及地区间的比较。

许多军事软件应用使用“挣值”来测量进度。一些民用的项目也使用这个方法，但是这种方法一般用在国防体系内。

2008 年左右，软件开发效率中有两个地方会用到功能点：其一是每个人每月对于功能点的贡献；其二是每个功能点开发耗费的工作时间。

软件的质量措施是鉴别顶尖软件开发商的有效途径。落后的软件企业几乎从来不使用质量措施，然而一流企业都会在这上面下工夫。质量措施包括收集源代码缺陷（需求、设计、代码以及不良修复）的数据量以及安全级别。

真正一流的企业也会度量软件的缺陷去除效率。这需要对软件的所有缺陷进行统计，这些缺陷包括开发期间以及产品发布后的所有缺陷。例如，一个公司在开发过程中发现了 900 个缺陷，然后产品发布后的三个内中发现了 100 个缺陷。那么该企业的缺陷去除效率就达到了 90% 的水平。一流软件公司的缺陷去除效率一般都在 95% 以上，这比美国软件公司的平均水平（85%）高出了 10 个百分点。

度量数据的另外一个用处就是与行业的基准进行比较。截至 2008 年年底，国际软件基准组织已经成了软件基准的主要来源，它拥有 4000 多个软件项目的数据。2009 年前后，一个最佳实践就是使用 ISBSG 的数据收集工具，该工具可以用来定期向 ISBSG 提供基准数据，包括需求到最后产品发布的所有数据。当然，机密应用和专有应用可能就不必如此了。

对于如何避免无效的或者违反标准经济学假设的度量措施和指标，一流的公司有很好的应对策略。以下罗列了两个常见的软件度量指标，它们违反了标准经济学的假定，因此不能用于经济分析：

- 平均缺陷成本指标不利于改进产品的质量，并且会使错误很多的软件看起来很不错。
- 代码行指标不利于高级语言，并且使汇编语言看起来更高效。

在本书后续的章节里，我们还将讨论这两个指标。代码行指标和平均缺陷成本指标都违反了标准经济假设，并且在生产率和质量上还会导致错误的结论。所以这两个度量指标都不适合经济研究。

目前在软件行业中，度量和指标做得不是很尽人意。一个原因是部分公司基本上不做度量，或者是很少做度量；另一个原因是，尽管一些公司使用度量措施，但是他们却使用

了无效的度量措施，如代码行度量或者平均缺陷成本度量。2009年，软件度量还是一个令人尴尬的职业，它急需在质量和质量的度量上做出重大改进。

在人类历史上的任何“工程”领域，软件行业里的度量措施应该算是最差劲的了。大部分的软件组织都不清楚如何与其他组织比较生产率和质量。历史数据的缺乏使得软件的评估变得相当困难，同时也使进度的改进成了天方夜谭，这也是为什么大型软件的失败率都很高。如果软件度量做得不好，那么这也应该算是专业失当。

2.31 软件的基准和基线

一个软件的基准是指，一个项目或组织与其他公司相似项目或组织的比较。一个软件的基线是指，收集特定时间内的质量和生产率信息。在软件过程改进期间，软件的基线用来评估过程情况。

尽管基准和基线有不同的用途，但它们收集各种类似的信息。所有的基准和基线收集的主要数据如下所示（但不限于以下信息）：

1. 行业编码，例如北美行业分类编码（NAIC）
2. 软件开发所在的国家 and 地区
3. 应用的分类，属性、范围、分类、类型
4. 应用程序的问题、数据以及编码的复杂度
5. 应用程序的规模（功能点）
6. 应用程序的规模（逻辑源代码）
7. 应用程序使用的编程语言
8. 应用程序所使用的可重用材料
9. 源代码语句与功能点的比例
10. 应用程序的开发方法（Agile、RUP、TSP等）
11. 应用程序所使用的项目管理和评估工具
12. 软件的能力成熟度模型集成
13. 开发活动账户列表
14. 活动级别的生产率数据（以功能点度量）
15. 总体净生产率（以功能点度量）
16. 成本数据（以功能点度量）
17. 项目所需的总人数
18. 项目所需的各类专家以及数量
19. 项目的总体规划
20. 开发、测试以及文档等活动的项目进度规划
21. 原始的潜在缺陷（需求、设计、编码、文档、不良修复）
22. 缺陷去除方法（审查、静态分析、测试）
23. 为应用程序设计测试用例

24. 缺陷去除效率水平

25. 项目开发过程中的延期和重大问题

我们所需要的不仅仅是以上 25 个专题的结果，还需要对这些结果进行回归分析。这样可以看出哪一种方法或工具对项目的影响程度最大。

咨询机构通常会收集以上 25 个方面的信息。问卷一般会包含 150 个具体的问题。对于拥有 10 000 个功能点的大型应用来说，通常需要两天的时间来收集所有的数据。这项工作一般由基准咨询团队现场进行。在数据收集阶段，一般会与项目经理和团队成员进行会话，以检验结果的有效性。访谈会议通常会持续两个小时，其中包括 6 个开发人员、专家以及项目经理等。

只有在少数非常专业的公司内，才会收集 25 个完整的基准和基线数据。多数公司只会用到其中的一部分基准，而这一般可以通过网络调查或者远程方式进行，而不需要现场访谈和数据收集。以下罗列了 10 个最常见的基准和基线（以降序排列）：

1. 应用程序的规模（功能点）
2. 应用程序所使用的可重用材料
3. 应用程序的开发方法（Agile、RUP、TSP 等）
4. 软件的能力成熟度模型集成
5. 总体净生产率（以功能点度量）
6. 成本数据（以功能点度量）
7. 项目所需的总人数
8. 项目的总体规划
9. 项目开发过程中的延期和重大问题
10. 客户报告的错误和缺陷

这部分的基准和基线当然是有用的，但是对所有影响应用结果的因素来说，这部分的基准和基线依然缺乏一个完整的静态分析。然而，一旦应用程序编写完成，这些基准数据可以通过远程的方式在两到三小时内收集完毕。

现场基准可以由内部人员完成，但是大多数情况下是由咨询公司来完成的，如 David 咨询集团、软件生产率研究所以及 Galorath 等公司。

在 2009 年的时候，主要的远程基准和基线机构是国际软件基准组织（ISBSG）。该组织是一个非营利的政府组织，总部设在澳大利亚。该组织已经收集了 5000 多个软件项目的数据，并且每年还会新增 500 多个项目。

尽管这样收集的数据没有现场收集的数据完整，但是这个方法依然可以说明总体的生产率。这个方法还能够显示出不同开发方法对软件所产生的影响，比如 Agile、RUP 以及类似的方法。然而，在写作本书的时候，软件的质量数据并不是很完整。

国际软件基准组织的数据是用功能点指标来表示的，这种方法是基准和基线的最佳实践。目前可以使用的功能点指标有 IFPUG 功能点指标、COSMIC 功能点指标以及其他几个变种，如芬兰功能点指标和荷兰功能点指标。

目前，国际软件基准组织倾向于信息技术以及网站应用，而有特殊用途的软件应用则

很少，如军事软件、嵌入式软件、系统软件以及科学应用软件等。当然，机密的军事应用软件是无法得到的。

在国际软件基准组织中，还存在另外一个缺陷，那就是统计功能点时存在不可避免的困难。因为功能点的分析非常繁琐，代价也相当昂贵，因此在 ISBSG 中，基本上没有那个项目的功能点超过 10 000 个。因此，ISBSG 缺乏大型操作系统以及企业资源规划包的数据，这些大型系统的功能点一般在 10 万到 30 万之间。

在 2009 年，出现了一些快速、低成本的功能点方法；但是由于它们是新出现的，因此它们还没有被运用在基线与基准的研究上。然而，到 2010 年或 2012 年，假设经济发展的情况下，那么这种局面将发生改变。

由于数据是客户自己远程提交给 ISBSG 的，因此尽管会纠正明显的错误，但是仍然没有一个正规的结果验证。在所有的自我分析检验中，或许会有因为误解或各地对如何测量执行不同标准而产生的错误。

国际软件标准组织有两个主要的优势：（1）它对大众开放；（2）它的数据量正在迅速增加。

应用软件的基线与基准被视为最佳实践，这是因为它们会阻止应用程序中非理性的规划需求，并且还会避免不成熟的管理和开发方法。

每一个大型项目开始的时候，都应该先借鉴一下 ISBSG 的基准信息。每一个过程改进计划都应该首先设定一个定量基线，并且该流程可以被度量。这两种最佳实践都应该在全世界范围内得到广泛应用。

2.32 软件项目的里程碑和成本跟踪

“里程碑跟踪”是指，每一个重要的可交付成果都有一个正规的结束标示。通常情况下，里程碑的结束是对可交付成果的审查的直接结果。项目里程碑不是一个任意的日历日期。

项目管理主要负责创建项目里程碑，监督项目里程碑的完成，并且准确地报告里程碑是否顺利完成。当遇到严重问题时，应该在项目里程碑完成前解决该问题。

对于拥有 10 000 个功能点的软件应用，一组典型的项目里程碑应该包括以下几个部分：

1. 需求审查
2. 项目规划审查
3. 成本及质量评估审查
4. 外部设计审查
5. 数据库设计审查
6. 内部设计审查
7. 质量规划以及测试计划审查
8. 文档规划审查
9. 部署规划审查
10. 培训规划审查

11. 代码审查

12. 每个开发测试阶段

13. 顾客认可度测试

一般情况下，失败或者延迟的项目通常都没有对项目的里程碑进行跟踪。当项目的活动还在进行中的时候，就报告该活动已经结束。项目里程碑可以是简单的日历记录，而不一定是实际的可交付成果的完整审查报告。有一些审查报告也会因为太过于简单而导致效率低下。报告偶尔也会遗漏一些东西，比如“培训”。

项目里程碑在软件行业里的意义是含混不清的，当然，能够认识到这一点已经算是难能可贵了。项目里程碑并不是指可交付成果的正式审查结果，而是指任意未经审查或未经测试的材料交付。

行业巨头一般不会提交不完整或者错误百出的文件和代码段作为里程碑，因为那样会影响后续的开发工作。

在里程碑跟踪中，行业巨头还应该注意另外一点，那就是当问题产生时，或者是项目出现延期时，到底应该采取什么措施。应对这些突发问题时，行动应当迅速敏捷，也就是说应该纠正行动规划，均衡任务分配，并且迅速纠正错误行动。另一方面，在一些稍差的公司里面，这些突发问题可能会被忽略，并且正确的纠正措施一般很少会出现。

许多案件中都会涉及失败的项目，在这种情况下，项目跟踪就似走马观花一样。问题一般都会被忽视或者搁置一旁，而不是得到重视，然后认真解决。

针对面向对象的项目，Shoulders 公司研发了一套有趣的项目跟踪机制。这种方法是使用规模不一的 3D 模型来表示软件的对象和类，这些模型的材料就是聚苯乙烯塑料球，最后通过销子将这些球连接成可移动模型。

这个模型最好放在一个显眼的位置，让尽可能多的团队成员都能看到。它的可移动性能够使得它随时向观看者进行展示。彩色的丝带说明了每个组件的状态，通过不同的颜色来表示设计完成、代码完成、文档完成以及测试完成（金色）。还有一些彩带用来显示可能出现的问题和延期。

这种方法使得整个项目的状态能够在瞬间显示。销售中通过使用 3D 模型软件包来实现自动化，但是在实际项目中，这种物理结构的优势更加能得到彰显。Shoulders 公司将大量重要的信息通过一种单一的可视化物体来呈现，这样一来，非技术人员就可以很容易看懂了。

2.33 软件发布前的变更控制

在分析和设计阶段，对于拥有 10 000 个功能点的应用来说，每月的需求变更率大约在 1% 到 3% 之间。一般情况下，需求阶段的功能点总数比部署阶段的功能点总数高出 50%。因此，对于拥有 10 000 个功能点的应用来说，成功的软件项目必须使用最先进的方法和工具，以确保变更不会失控。对于成功的项目来说，具体变更的需求一般是由变更委员会来进行评估。当然，所有对成本和开发进度产生重大影响的变更都会引发项目的升级计划和新的成本估算。

对于拥有 10 000 个功能点的应用来说,最先进的变更控制技术包括:

- ☐ 在审批变更时,指定关键可交付成果的“责任人”。
- ☐ 锁定所有可交付成果的“主版本”,所有的变更只能通过正规途径。
- ☐ 规划多个版本的内容,并且为每一个版本指定关键的功能。
- ☐ 在项目开始之前,评估开发变更的速度以及数量。
- ☐ 使用功能点指标来量化变更。
- ☐ 一个联合客户和开发的变更控制委员会或者指定的领域专家。
- ☐ 使用联合应用设计(JAD)以减少后续的变更。
- ☐ 使用正规的要求审查以减少后续的变更。
- ☐ 使用正规的原型来减少后续的变更。
- ☐ 使用迭代开发计划以适应变更。
- ☐ 使用敏捷开发计划以适应变更。
- ☐ 对所有变更请求进行正规的审查。
- ☐ 对于所有多于 10 个功能点的变更,需要对成本和进度的评估进行修订。
- ☐ 优先考虑对业务有影响的变更请求。
- ☐ 在具体版本上分配正式变更请求。
- ☐ 结合交叉引用来使用自动变更控制工具。

使用正规的 JAD(联合应用设计)会有一个明显好处,那就是可以减少后续的需求变更。

IBM 以及其他一些公司对 JAD 进行了研究,研究表明由于 JAD 技术的高效性,可以使未经规划的需求变更每月减少到 1% 以下,而不是预期的 1% 到 3%。

使用原型也有助于减少后续的变更需求。通常情况下,关键界面、输入和输出都被原型化,这样一来用户就能获得一些实践经验,并且认识到完成后的应用程序看起来应该是怎样的。

然而,变化总在大型系统内发生。想要冻结现实世界中任何应用的需求是不可能的,如果真想这么做那也未必太天真了。因此,面对变更,一流的公司一般都会沉着应战,而不是让这些变更阻碍项目的进度。所以说某种形式的迭代开发在逻辑上是十分必要的。

新的敏捷方法能够适应需求变更。这种方法是在一个开发团队中设置一个固定用户代表。这种方法能够以最快的速度构建项目的基础功能,然后根据用户的实际体验来收集新的需求。

这种方法适用于只有少数用户的小型项目。它还没有被应用于诸如微软、Vista 这类用户数量和应用特征数以百万千万计的项目上。对于这样大规模的项目来说,一个用户甚至一个小团队的用户都不可能反映出整个使用范围的模型。

有效的软件变更管理是一个复杂的多维度问题。软件的需求、计划、规格、源代码、测试计划、测试用例、用户手册以及许多其他类型的文档经常会发生变更。这种变更也许由外部因素引起,如政府法规、竞争因素、新业务需求、新技术或者漏洞修复等。

此外,一些变更通常会波及许多不同的可交付成果。比如一个新需求就可能会导致需求文档的变更,还会引起内部和外部规格、源代码、用户手册、测试库、开发计划以及成

本核算的变更。

在需求结束和产品交付后，对功能点分别进行度量，我们会发现，在设计和编码阶段，“需求蔓延”的增长速度大约为每月 1% 到 3% 左右。所以，对于拥有 1000 个功能点的应用来说，在需求结束之后，在随后的 8 到 10 月中，功能点会以每月 20 个的速度增长。最大增长幅度可以达到每月 5% 的速度，最小增幅是每月 0.5%。当然，快捷开发的增幅是最高的。

对于一些政府和军事项目来说，其中一个要求就是可追溯性。这意味着说，所有代码或可交付成果的变更都必须能够追溯到一个明确的审批，这个审批由政府项目干系人或发起人负责。

除此之外，有一部分人可以被授权更改相同的部分，例如源代码模块、测试用例以及规格。很显然，单独的变更必须要协调一致，因此“锁住”原版副本的关键可交付成果，并且序列化各种来源的更新就是非常必要的。

由于变更控制比较复杂并且覆盖面广，因此许多可使用的自动工具都不能帮助保持当前变更，而且还能应对多个可交付成果的变更。

然而，变更控制是不能够完全自动化的，因为，对于应用程序的范围、新需求以及可能引起项目进度和成本变化的重大变更，股东、开发人员以及其他关键人员的审批是必需的。

在某种程度上，源代码的各种变更会产生新的测试用例。正规集成和新构建将会带来一系列的变更。这些构建可能在有需要时出现，或者在固定的时间间隔出现，如每天或每周。

如果在程序开发期间内，变更控制处理不当，通常会引起很多不必要的麻烦。这一点我们将在后续的章节进行讨论。在软件的维护期间，我们也要处理好部署问题。变更控制是一种超级配置控制，因为变更控制也包括企业决策的竞争力和信誉，这些都是配置控制以外的一些东西。

2.34 配置控制

通常来说，配置控制是变更控制的一个子集。正规的配置控制起源于 20 世纪 50 年代，最早是在美国国防部兴起，该方法一般用来跟踪复杂武器的部件和演化系统。换句话说，硬件的配置控制比软件的配置控制资历更老。一般来说，配置控制是一种机械活动，它由许多工具以及自动化功能支持。配置控制涉及跟踪成千上万个文档以及源代码的更新。确定一个特定的变更是否有价值，并不是配置控制范围之内的事情。

软件配置控制是 SEI（软件工程研究所）能力成熟度模型（CMM/CMMI）的“关键实践”领域之一。也有其他一些标准组织涉及配置控制，这些组织包括：IEEE（美国电子电气工程师协会）、ANSI（美国国家标准协会）、ISO（国际标准化组织）等。例如，ISO 标准 10007-2003 以及 IEEE 标准 828-1998，这两个标准都涵盖软件应用的配置控制。

虽然配置控制大部分是自动化的，但它仍然需要人工干预才能发挥最大作用。显然，功能需要被唯一标示，而且在需求、规格说明、源代码、测试用例以及用户文档中要有广泛的映射。这样就能确保每个（影响多个可交付成果的）特定功能的变更能够正确连接到相关的可交付成果上去。

此外,必须对每个交付成果的原版副本上锁,以避免意外变更的发生。只有正规的验证才可以更新可交付成果的原版副本。

对于所有想给客户实际版本的应用来说,自动配置控制是最佳的实践。

2.35 软件质量保证

软件质量保证是指,在应用交付给客户之前,组织和专家通过各种手段来确保和证明软件应用的质量水平的过程。

在一些大企业中,SQA组织是独立于软件开发组织的,如IBM,该组织只需要向公司负责质量监管的副总裁进行报告。这样做是为了确保没有外在的压力或职业威胁来影响SQA人员做出错误评估。

在软件工程项目中,SQA人员的数量应占到整个项目人员的3%~5%。如果SQA人员的比例低于3%,那么这个项目的SQA人员数量应该是不够的。如果是通过“令牌轮询SQA”组织进行审查,那么其人员的严重不足会导致他们无法完成对交付成果的审查,因此这种做法也是不值得提倡的。

软件质量保证组织应该由一些训练有素的人员组成,这些人员都学习过质量检测和质量控制。许多软件质量保证人员都通过了六西格玛黑带的认证。软件质量保证组织不是只负责测试的组织,并且也不可能不做任何测试。通常情况下,软件质量保证组织的基本职能有以下几个:

- 评估质量水平方面的潜在缺陷以及缺陷去除效率。
- 度量缺陷去除效率,并指定缺陷的严重级别。
- 在软件应用中使用六西格玛方法。
- 在软件应用中使用质量功能展开(QFD)的方法。
- 主持并参与正式审查。
- 教授质量控制方面的话题。
- 监督相关企业是否遵守ANSI、ISO质量标准。
- 审查所有可交付成果,确保遵循质量标准和实践。
- 审核测试计划和质量计划,确保项目完整性以及最佳实践方法。
- 测试度量结果。
- 分析严重缺陷的根源。
- 向级别更高的管理者报告潜在的质量问题。
- 通过颁发质量水平认证书来认可向客户交付的产品。

在IBM以及其他一些公司,在应用程序交付给客户之前,有一个必需的环节,那就是需要软件质量保证组织发布一个正式的审核文件。如果由于产品的质量问题的,软件质量保证组织不批准产品交付,那么这种决定只能由公司的副总裁或公司总裁来推翻,其他人无权干涉。通常情况下,质量问题是不可逆的。

软件质量保证组织是度量软件质量的主要单位,他们要明确的质量问题主要有以下几

个方面(但不限于以下这些)。

顾客满意度:很多一流公司都会进行年度或者半年度的客户满意度调查,以收集客户对其产品的看法。这些公司还会通过互联网来获取详细的缺陷报告以及客户支持信息。在商用软件圈内,许多一流的企业都拥有活跃的用户群和论坛。这些组织经常会对质量和满意度等话题开展独立的调查。有的也有焦点团体,并且有些软件公司甚至还设有正规的可用性实验室。在这个实验室,新版本的产品在受控制条件下接受客户的试用。(注:客户满意度通常是由市场营销组织调查而不是由软件质量保证组织来调查。)

缺陷的数量和来源:行业内的一流企业都会对主要可交付成果的错误和缺陷保持准确的记录,他们从软件的需求和设计阶段就开始这么做了。至少有5个类别的缺陷需要检测:需求缺陷、设计缺陷、代码缺陷、文档缺陷以及不良修复缺陷,还有就是在修复一个缺陷时引入的次生缺陷。准确的缺陷报告是改进产品质量的关键因素。实际上,通过分析缺陷数据来寻找导致缺陷的根本原因,是很多公司实行预防缺陷以及移除缺陷的新方法。总的来说,对缺陷进行详细的度量并对后续数据进行细致的分析是软件公司性价比最高的活动之一。

缺陷去除效率:对于不同类型、不同规模的项目,行业巨头们熟知每一种主要检查、审查、测试的平均效率和最佳效率,因此可以选择出一系列最佳的缺陷去除步骤。由于单独测试已经不是很有效了,因此在美国波多里奇(Baldrige)国家质量奖的获奖者和组织中,使用预备测试审查是大家常用的做法。在产品交付给客户前,他们能够使缺陷去除效率达到95%,甚至是99%,而行业的落后者则很少能达到80%,有时甚至可能低于50%。

应用程序中交付的缺陷:当软件交付给客户之后,行业巨头们就会开始统计客户报告的错误。然后,向主管提交月度报告,这里面要显示所有产品的缺陷趋势。这些报告还会以年度总结的形式出现,这其中也包括一些补充的缺陷统计,这些统计一般由国家、州、行业以及客户等报告。

缺陷的严重性等级:毫无疑问,几乎所有的行业巨头都会对错误和缺陷的严重性进行评估。缺陷严重性分为5个等级,从1级到5级。一般来说,1级缺陷会导致系统完全崩溃,随后的等级严重程度依次递减。

软件的复杂性:众所周知,复杂的代码是很难维护的,而且缺陷率比平均水平要高出很多。市面上有各种各样的复杂性分析工具,它们支持标准复杂性的度量,如循环复杂度以及基本复杂度。有趣的是,系统软件社区比信息技术(IT)社区更容易度量复杂性。

测试用例的覆盖率:软件测试可能会覆盖也可能不会覆盖应用程序中的每一个分支和路径。各种各样的商业工具能够监控软件的测试结果,并且能够帮助确定应用程序测试过程中的疏漏。那么在这一方面,系统软件领域比信息技术(IT)领域更容易度量测试覆盖率。

质量控制和缺陷修复的成本:质量度量的一个重要方面是,准确地记录所有与缺陷预防和缺陷去除相关的资源和成本。对于软件来说,这些度量包括以下成本:(1)软件评估;(2)质量基线研究;(3)检查、审查和测试;(4)保修期内的修复以及产品发布后的维护;(5)质量工具;(6)质量教育;(7)软件质量保证组织;(8)用户满意度调查;(9)任何涉及质量的诉讼或者由于质量低劣导致的用户的流失。总的来说,Crosby的“质量成本”

原则还是能够应用于软件的，但是大多数的公司都扩展了该原则的基本概念，并且跟踪了软件项目相关的一些额外因素。质量成本的一般主题包括：预防成本、评估、内部故障以及外部故障等。对于软件来说，由于某些特殊的需求，比如毒性需求（Toxic Requirements）、安全漏洞以及性能问题等，这些问题都不能够通过正规的生产质量成本来处理，所以需要有更多的细节。

质量的经济价值：在软件的质量保证中，质量的经济价值是很少涉及的。美国国际电话电信公司（ITT）的前任质量副总裁 Phil Crosby 曾经说过一句经典的话，“质量是免费的。”但是对于软件行业来说，质量最好不是免费的。它带来的效益远大于它自己本身。在已交付成果中，每减少 120 个缺陷就能够减少一个维护人员，每减少 240 个缺陷就能够减少一个客户支持人员。用今天的话来说，软件工程师每年可以花更多的时间来修复系统漏洞，而不是再做实际的开发工作了。在软件的整个生命周期中，以质量为中心的开发方法都可以降低成本并且缩短开发时间，如团队软件过程（TSP）、联合应用设计（JAD）、质量功能展开（QFD）、静态分析、审查以及测试等。不幸的是，除非是在非常先进的公司，不然的话，想要使贫乏的度量实践去改进项目的成本或时间，那简直是天方夜谭。

如前所述，一个关键的原因就是两个常用的质量度量标准（代码行和平均缺陷成本）。这两个标准本身就有缺陷，并且不能够用来处理经济问题。在每个功能点上使用缺陷成本去除法是一个很好的选择；但是这些标准需要在组织中使用，并且组织能够同时统计工作量、成本以及质量方面的数据。由笔者的研究来看，综合使用缺陷预防和缺陷去除方法可以降低潜在缺陷，并且将缺陷去除效率提高到 95%。同时还能使开发成本、开发进度、维护成本以及客户支持成本都从中受益。

在任何形式的软件度量中，质量度量几乎是最重要的。这是因为糟糕的质量总是会导致进度延误和成本超支，而卓越的品质往往与软件应用的按期完成以及有效的成本控制息息相关。

在构建大型复杂的物理设备时，我们经常会看到正规 SQA（软件质量保证）组织的身影，这些大型设备包括飞机、大型机、电话交换系统、军事设备以及医用设备等。这样的机构已经清楚地认识到了一点，那就是质量控制是成功的关键。

相比之下，有些组织构建信息技术软件时可能就没有 SQA 组织，如银行和保险公司。如果他们这样做，那么这个组织虽然会负责测试，但是他们并没有执行一个完整的质量活动。

通过对可交付软件应用的质量进行研究，我们发现，有些公司的累计缺陷去除效率竟然可以超过 95%；不过这些公司有一个共同点，那就是它们拥有正规的软件质量保证组织和正式测试组织。

2.36 审查以及静态分析

IBM 公司早在 35 年前就已经开始使用正规的设计和代码审查了。他们在缺陷去除率上仍然是顶级的。（前 IBM 金斯顿的雇员 Michael Fagan 和他的同事 Lew Priven、Ron Radice、Roger Stewart 首次推出了审查方法。）此外，审查与其他形式的缺陷去除方法（如

测试和静态分析)有协同关系,并且这些缺陷预防的方法都是相当成功的。

自动静态分析是一种新式的技术,该技术起源于 12 年前。自动静态分析会检查源代码的句法错误,当然也会检查边界条件、调用、链接等一些其他棘手的错误。静态分析可能不会找到嵌入的需求错误,如臭名昭著的 Y2K^①问题,但是它确实是非常有效的方法,它可以在源代码中找到成千上万的错误。审查和静态分析是协同的缺陷去除方法。

Tom Gilb 是一位软件审查方面的著名作家。最近,他和他的同事们结合他自己一个作品提出了一个观点。他们认为,人类的思维是能够查找和消除复杂问题的最佳工具,这些问题可能源于需求、设计以及其他非代码的可交付成果。事实上,对于在源代码中寻找更深层次的问题,正规的代码检查在缺陷去除效率级别上优于测试。然而,在寻找越来越广泛的问题上,静态测试和自动化测试却是相当有效的方法。

如果一个应用程序所使用的编程语言支持静态分析(如 Java、C、C++ 等),那么静态分析就是最佳实践。静态分析也许可以发现 87% 的常规编码缺陷。不过,偶尔也会有误报。但是,这些误报可以通过“微调”静态分析工具(匹配具体应用)来减少。静态分析后的代码检查能够发现一些更深层次的问题,例如需求中嵌入的缺陷,特别是在关键的模块和算法上。

因为无论是代码审查还是静态分析,它们都不能够在运行过程中很好地发现性能问题,因此必须结合使用动态分析工具,要么使用各种受控的性能测试套件,要么通过检测应用程序自身以记录应用程序时间和性能方面的数据。

在寻找错误或缺陷上,多数测试的效率低于 35%。正规的设计检查和代码审查的平均缺陷去除效率超过了 65%,这大约是多数测试方法效率的两倍。还有一些审查方法的缺陷去除效率达到了 85%。Tom Gilb 曾经在报告中指出,有些方法的审查效率竟然高达 88%。

将正规的需求审查和设计、静态分析、正规的专家测试以及正规的软件质量保证组织相结合,能够使项目的累计缺陷去除效率超过 99%。

正规的审查是一个手工活动,该团队一般由 3~6 个人组成,他们会根据一个正规的协议,一页一页地去审阅设计规范。一般情况下,这项工作有 4 个角色,分别是:主持人、记录员、审阅人以及一个打杂的。(有时候,新员工或者专家也会参与。)代码审查也是一样的,但是他们会逐行审阅屏幕和列表上的代码。这个活动之所以被称之为审查,那是因为代码或规范必须达到某一标准,这些标准包括(但不限于)以下几点:

- 必须有一个主持人保证活动进行顺畅。
- 必须有一个记录员负责记笔记。
- 在每个会议前,必须有充足的准备时间。
- 记录中必须详细记录发现的缺陷。
- 缺陷数据不应该用于考核或惩罚。

审查最原始的概念是以实际的会议为基础的,而该会议必须有现场的参与者。高速的网络通信以及支持远程审查的工具的出现,意味着审查可以通过电子方式来进行。这样一

① Y2K 是“Year 2000”的缩写,即 2000 年,有时也称千禧危机、千年虫、千年问题。——译者注

来就可以节省由于地理位置的分散而给团队带来的差旅费。

任何可交付的软件都可以接受正规的审查。下面的可交付成果已经拥有了足够的历史数据来表明整个检查过程是有益的：

- 构架审查
- 需求审查
- 设计审查
- 数据库设计审查
- 代码审查
- 测试计划审查
- 测试用例审查
- 用户文档审查

对于每一个使用正规审查的软件构件，在缺陷去除效率上，审查的范围是从 50% 以下到 80% 以上，并且平均去除率水平大约在 65% 左右。对于任何已知形式的缺陷去除，这一水平总体来说已经是最好的了。

此外，由于人类思维的灵活性以及处理归纳逻辑和演绎逻辑的能力，因此审查也是最常用的缺陷去除方法，并且基本上可以应用于任何软件构件。实际上，为了微调审查过程并消除瓶颈和障碍，审查甚至还被应用于其本身。

有的时候我们会被问到“既然审查的方法这么好，那么为什么这种方法没有被广泛使用？”这个问题的答案揭示了软件行业的一个弱点。在公共领域，软件审查已经有 35 年的历史了。除了少数几个培训公司尝试推销自己的“审查”技术外，其他所有的供应商都在推销自己的“测试”工具。因此，如果你想要使用审查技术，那么你就得先找到他们，然后才能使用。

针对有效的工具和技术，大多数的软件开发组织实际上并没有做专门的研究或者数据收集。在很大程度上，他们决定选择哪种工具和技术取决于供应商的言辞，他们一般会采用那些最有说服力的销售人员所推荐的产品。更离奇的是，销售人员会把他们的产品比作成银弹，一旦部署就可以产生意想不到的效果，并且几乎不需要任何培训、准备以及任何额外的努力。由于审查技术是不通过工具供应商来出售的，同时学会该技术还得投入许多精力并且参加培训，因此这项技术听起来就不是那么诱人了。所以，很多的软件组织甚至不知道什么是审查技术，并且也对其通用性和有效性知之甚少。

有一个很常见的做法是，所有顶级的软件质量组织，甚至还有一些美国的企业都倾向于使用预测试审查技术。许多大型制造商都对正规的审查技术情有独钟，这些制造商包括电脑制造商、电信运营商、航空航天、国防制造商、医疗器械制造商、软件系统以及操作系统等。所有这些都需要高品质的软件才能营销他们的产品，并且审查技术在缺陷移除方法中算是出类拔萃的了。

应该尽量避免毒性需求、需求错误以及需求遗漏等情况的出现，并且还得确保这些糟糕的情况没有影响到编码，因为需求中的问题很难通过测试发现。

虽然说测试能够检查出设计中存在的问题，但是最好在编码前就能检查出设计的问题。

最关键的是，在缺陷产生后，我们要立即响应，争取在几小时或者几天内解决它们。在特定阶段（如需求阶段）产生的缺陷，就应该避免其进入设计和编码阶段。

在一个特定的阶段或最短的时间间隔里，当缺陷出现时，可以使用以下罗列的已知的最有效的缺陷去除方法：

从表中可以看到，审查并不是唯一的缺陷去除方法，但在审查需求缺陷的时候，审查确实是最有效的方法，并且在查找其他缺陷时，审查也是相当有效的。

一个非营利组织于 2009 年成立了，该组织致力于提供正规审查相关的指导和量化数据。这个组织在本书写成时已经开始正常工作了。

截至 2009 年，很多工具都支持审查技术，这些工具可以对缺陷、缺陷去除效率、成本以及其他相关的因素进行预测。除此之外，这些工具还能够用来收集缺陷和工作量上的数据，并将其与静态分析和测试中的相似数据整合在一起。对于软件应用中所有的关键任务来说，正规审查就是最佳实践。

缺陷来源	最佳的缺陷发现方法
需求缺陷	正规的需求审查
设计缺陷	正规的设计审查
代码缺陷	静态分析 正规的代码审查 测试
文档缺陷	文档编辑 正规的文档审查
不良修复	缺陷修复后，再重新审查 缺陷修复后，重新运行静态分析工具 回归测试
测试用例缺陷	测试用例审查

2.37 测试和测试库的控制

自 60 多年前软件开始出现时，测试就已经成缺陷去除的主要方法了。目前，至少有 20 多种测试方法，并且有 3 到 12 种代表性的测试方法几乎被应用到了所有的软件。

需要注意的是，测试也可以与其他的缺陷去除方法结合使用，例如静态分析和正规的审查。实际上，像这样的协同组合就是最佳实践，因为就测试本身来说，它根本没法实现很高的缺陷去除效率。

不幸的是，在度量时，测试的缺陷去除效率水平很低。在缺陷去除效率上，很多形式的测试甚至低于 35%，比如单元测试，或者只能找到 1/3 的缺陷。

各种测试的累计缺陷去除效率很少会超过 80%，因此需要额外的步骤（如审查和静态分析）来提高缺陷的去除率，并使缺陷的去除率达到 95% 以上，当然这仅仅是一个最低的安全标准。

由于测试的形式很多，并且测试用例也多如牛毛，因此测试库自然非常巨大、烦琐，并且成功的管理往往都需要自动化控制。

测试的种类很多，包括黑盒测试（不知道应用程序的内部结构）、白盒测试（知道应用程序的内部结构）以及灰盒测试（知道应用程序的数据结构）等。

测试的另一种分类方法是根据测试的步骤来划分的，每一个步骤分别由开发人员、测试专家或质量保证人员以及客户自己执行。所有测试总的工作量大约是软件开发总工作量的 20% ~ 40%。由于测试的缺陷去除效率很低，因此可以考虑结合成本较低但去除率较高

的方法。

也有非常专业的测试方法，如测试性能、安全性以及实用性相关的问题。高安全标准的应用也可以通过能够突破应用程序防御系统的专业黑客来测试，所以这里不是通常意义下的“测试”。常见软件测试的方法包括：

开发人员的测试

子程序测试

模块测试

单元测试

测试专家或软件质量保证组织的测试

新功能测试

组件测试

回归测试

性能测试

安全测试

病毒以及间谍软件测试

可用性测试

可伸缩性测试

标准测试（确保遵循 ISO 以及其他标准）

本地化测试（外语版本）

平台测试（替代硬件或操作系统版本）

独立测试（军事应用）

组件测试

系统测试

客户或用户的测试

外部 beta 测试（商业软件）

验收测试（信息技术；外包应用）

内部客户测试（特殊硬件设备）

近年来，自动化促进了测试用例的开发、测试脚本的开发、测试执行以及测试库的管理。然而在开发测试计划、测试用例以及测试脚本上，人类的智慧仍然是非常重要的。

在文献中，一些问题被忽略了，应该深入地研究一下才行。其中之一就是测试用例本身的错误密度。从 IBM 的测试库中选择样本进行研究，我们发现有些测试用例中的错误竟然比软件中的错误更多。另一个问题是，测试中会有冗余的用例，这意味着两个或多个测试用例是重复的，或者不同的测试用例测试了相同的条件。显然，这是由于不严谨造成的，并且还会增加测试成本。这种情况通常是由于多个开发人员或多个测试人员测试相同的软件造成的。

这个论证已经被研究过，但需要投入更多研究的是测试缺陷去除效率。由于大多数测试方法的缺陷去除效率似乎都低于 35%，或只能找到 1/3 的缺陷，因此目前迫切需要知道

为什么会这样。

一个相关的话题就是，当使用各种测试覆盖分析工具时，我们发现测试的覆盖率很低。通常情况下，在测试过程中，应用程序只有 75% 或者更少的源代码会被执行。其中有一些可能是死代码（这是另外一个问题），还有一些可能路径很偏，但是有些则可能是碰巧遗漏的代码段。

最根本的是，测试本身不足以实现 95% 或者更高的缺陷去除效率。目前最好的做法是将测试与其他方法相结合使用，这些方法包括需求和设计审查、静态分析以及测试之前的代码审查。缺陷预防和缺陷去除应该以协同的方式结合进行。

有效的软件质量控制是区分成功项目与失败项目的关键因素。这是因为，对于大系统来说，寻找和修复漏洞的成本是最高的，并且耗费的时间也多于其他的任何活动。

成功的质量控制包括缺陷预防、缺陷去除以及缺陷度量活动。缺陷预防阶段的首要目的是将产生错误和缺陷的概率降到最低。缺陷预防活动包括六西格玛法、联合应用设计（JAD）（为收集需求）、正规设计方法的使用、结构化编码技术的使用以及可重用材料库的使用。

缺陷去除阶段的主要任务是发现可交付成果中所有的错误和缺陷。这个阶段的活动有需求审查、设计审查、文档审查、代码审查以及所有形式的测试。以下罗列了 2009 年较为重要的缺陷预防和缺陷去除活动：

缺陷预防

- ☐ 使用联合应用设计（JAD）来收集需求。
- ☐ 使用质量功能展开（QFD）来获得质量需求。
- ☐ 正规的设计方法。
- ☐ 结构化编码方法。
- ☐ 在对遗留代码更新之前，先对代码进行改造。
- ☐ 在对遗留代码更新之前，先分析代码的复杂性。
- ☐ 从遗留代码中移除易错模块。
- ☐ 正规的缺陷和质量评估。
- ☐ 正规的安全规划。
- ☐ 正规的测试计划。
- ☐ 正规的测试用例结构。
- ☐ 正规的变更管理办法。
- ☐ 六西格玛方法（用于定制软件）。
- ☐ 利用 SEI（软件工程研究所）的 CMM（能力成熟度模型）或 CMMI（能力成熟度模型集成）。
- ☐ 使用新团队和 PSP（个体软件过程）（TSP、PSP）。
- ☐ 用户参与开发团队工作（如敏捷方法）。
- ☐ 在编码前创建测试用例（如极限编程）。
- ☐ Scrum 每日会议。

缺陷移除

- ☐ 需求审查
- ☐ 设计审查
- ☐ 文档审查
- ☐ 正式安全审查
- ☐ 代码审查
- ☐ 测试计划和测试用例的审查
- ☐ 缺陷修复审查
- ☐ 软件质量保证评估
- ☐ 自动软件静态分析（对于 Java 或者 C 语言适用）
- ☐ 单元测试（手动或自动）
- ☐ 组件测试
- ☐ 新功能测试
- ☐ 回归测试
- ☐ 性能测试
- ☐ 系统测试
- ☐ 安全漏洞测试
- ☐ 验收测试

在软件的总缺陷数上，缺陷预防和缺陷去除活动的结合会导致成功的项目与失败的项目有很大的差异。对于拥有 10 000 个功能点的软件项目来说，成功项目中的累计开发缺陷一般为 4 个缺陷 / 功能点，并且在产品交付前，缺陷去除效率可以达到 95%。换句话说，已交付成果中的缺陷数量大约是 0.2 个缺陷 / 功能点，而总的缺陷个数为 2000 个。在这 2000 个缺陷中，大约有 200 个缺陷（或 10%）属于严重缺陷。其余的则都是无关紧要的缺陷。

相比之下，失败项目中的累计开发缺陷一般为 7 个缺陷 / 功能点，而在交付产品前，缺陷去除效率仅有 80%。交付产品中的缺陷数量大约是 1.4 个缺陷 / 功能点，也就是说总共有 14 000 个潜在缺陷。在这 14 000 个缺陷中，大约有 2800 个缺陷（20%）属于严重缺陷。在交付给客户后，这大量的潜在缺陷会引起很大的麻烦。

失败的项目通常都忽略了设计、代码审查以及静态分析，并且完全依赖于测试。前期对审查的忽略会导致三个严重的问题：（1）当测试已经减缓了项目的进度，并使项目趋于停滞状态时，项目中仍然有大量的缺陷；（2）对于没经审查的项目来说，“不良修复”的注入率高得惊人；（3）只使用与测试相关的方法，很难使总的缺陷去除效率达到 80%。

2.38 软件的安全性分析与控制

在本书出版之时，软件安全正日益成为一个重要的话题。不仅是个别的黑客试图闯入电脑和软件应用程序，也有有组织的犯罪、贩毒集团、恐怖组织（如“基地”组织），甚至还有敌对的外国政府。

随着计算机和软件越来越普遍地应用于商业和政府运作，财务数据、军事数据、医用数据以及警务数据的价值都是非常高的，因此犯罪分子会利用复杂的工具和专业的黑客来进行主要的攻击。网络安全正成为一个主要战场，因此我们需要认真对待。

当前，包含敏感数据（如金融信息、医疗记录、人员数据以及军事和机密信息）的软件应用每天都存在着各种各样的风险，这些风险可能来自黑客、病毒、间谍软件，甚至可能是心怀不满而故意盗窃的员工。软件应用的安全控制是一件非常严肃的事情，它与主要的成本和风险密切相关。安全控制做得不好可能会导致严重的损失，并且如果软件管理者和企业高管没有执行更高级别的安全控制的话，那么他们可能会面临刑事指控。

当前，关键软件应用的安全控制需要与专业技能、复杂的软件工具、适当的架构、设计、编码实践以及高度的警惕性相结合。附加的工具和方法也是必需的，如硬件安全设备、电子监控、详细调查所有人的背景等。除此之外，还可以雇用职业黑客，让他们专门查找软件的弱点和漏洞。

然而，软件的安全是以严谨的架构、设计以及编码实践为基础的。此外，使用安全审查和安全专家是优秀的安全控制的关键标准。应用程序的“黑名单”和“白名单”与应用程序的接口需要进行安全分析。还有一点就是，以安全为核心的编程语言（如 E 语言）也是非常重要的，这也是一种最佳实践。

安全漏洞往往有不同的来源，包括用户输入、应用程序接口等，当然漏洞往往是由于糟糕的错误处理或编码实践造成的。由于软件工程师在安全方面的培训不够充足，因此在减少安全漏洞上，专家是必需的。

目前，活跃在安全领域的公司有几十个。美国国土安全部打算建立一个新的研究室专门应对软件安全。非营利组织（如资讯安全中心，CIS）的会员数量正在快速增加，对企业和政府机构来说，加入这样一个组织将是一个最佳实践。

此外，安全标准（如 ISO 17799）也为软件安全主题提供了指导。

尽管黑客和网络盗窃是最普遍的安全问题形式，但计算机和数据中心的物理安全也是非常重要的。几乎每个月都会出现信用卡和医疗记录丢失的报道，而原因则是笔记本电脑或台式电脑被盗。

由于物理盗窃和黑客攻击已经司空见惯了，因此对有价值的信息（如专有信息和机密信息）进行加密自然成了一个最佳实践了。

自 2000 年以来，黑客和安全专家之间已经开始了角逐。不幸的是，道高一尺魔高一丈，黑客正变得越来越精明，并且数量也越来越多。

在理论上，我们是能够构建某种形式的人工智能或神经网络安全分析工具的，该工具能够检查软件应用并发现其中的安全漏洞。事实上，一个类似的人工智能工具可以应用到架构和设计中，并且为架构师和设计师提供最安全的解决方案。

在软件应用的开发中，一套最佳实践包括：

□ 在安全标准方面，改善本科生和软件工程师的专业培训。

□ 对于每一个必须要连接到互联网或其他电脑的应用程序，我们必须开发一个正规的安全计划。

- 对需求和规范执行安全审查。
- 确保开发团队的人身安全。
- 为家庭办公室和便携式设备提供顶级的安全标准。
- 使用高度安全的编程语言（如 E 语言）。
- 对代码执行自动静态分析以发现潜在的安全漏洞。
- 对正在更新的遗留应用执行静态分析。

未来，自动化可能将成为最佳实践。然而，截至 2009 年，专家的安全分析依然是最佳实践。不过，安全专家一般都出现在军事应用和机密应用中，而在民用应用中则很少见到他们。

2.39 软件的性能分析

Windows XP 或 Vista 的用户可以观察到，大型复杂软件应用的性能并不像想象中那么复杂。拿 Windows 举例来说，随着时间的推移，应用程序加载的时长也会变长。越来越多的互联网与间谍软件的结合，致使系统的运行速度略有下降。

虽然一些实用的应用程序可以恢复原来的性能，但事实上，性能优化仍然是亟待提高的一项技术。性能迟缓的软件不仅只在微软公司出现。各种 Symantec 的工具都遭到了频繁的投诉，比如诺顿的防毒软件，它的速度很让人着急。据笔者个人观察，假如电脑没有安装最新的 CPU 的话，诺顿杀毒软件在 24 小时内是无法完成扫描的。

因为性能分析并不总是软件工程或计算机科学课程里的一部分，所以许多软件工程师都不能很好地处理性能优化的问题。在像 IBM 这样的大公司，他们会雇用一些专业人员，这些专业人员在处理软件性能方面接受过特殊的训练。假如公司开发功能点大于 10 000 的软件应用，这类专家当然是最佳人选。

目前，有许多性能工具和度量设备，比如收集动态数据的分析器。当然，将软件性能的度量能力嵌入到软件应用本身也是可行的，这被称为“插桩技术^①”。

因为插桩技术和其他形式的性能分析工具可能会降低程序运行速度，所以需要必要的维护来确保数据的正确性。性能优化领域用到了几个物理学方面的术语，例如，“海森堡漏洞”（Heisenbug）是根据海森堡不确定性原理命名的，它是指每次在调试分析时就消失的漏洞。另一个物理学术语是“波尔漏洞”（Bohrbug），它是根据尼尔斯·玻尔的名字来命名的，这种漏洞只有在特殊的条件下才会出现。第三个物理学术语是“曼德尔漏洞”（Mandelbug），它是以伯尼特·曼德尔布罗特的名字来命名的，他创造了混沌理论，这种形式的漏洞是由随机的和混乱的因素引起的，所以隔离起来很困难。第四个漏洞是“施罗德漏洞”（Schrodengbug），该漏洞非常少见，是以恩斯特·施罗德的名字命名的。通常情况下，这种漏洞是不会出现的，除非有人意识到那些代码根本就不应该执行；并且据说，当这些漏洞被发现时，软件也会停止工作。

① 插桩技术是指，在保证被测程序原有逻辑完整性的前提下，在程序中插入一些探针（又称为“探测仪”），通过探针的执行并抛出程序运行的特征数据，来对应用程序的性能进行分析。

性能问题的出现也基于商业周期。例如，许多的金融和会计软件包会在一个季度或者一个财政年度的使用后，运行速度会显著变慢，那是因为使用量的骤增造成的。

软件在运行过程中，如果严重的漏洞导致软件无法运行，那么软件的性能将为零，而在性能这一章中没有对其很好的讨论。这种问题能够通过平均失效时间来测量。这样的问题在软件交付后一到两个月内是很常见的，但是会随着软件的逐步稳定而减少。拒绝服务攻击也会使软件停止工作，这种情况越来越常见了。

最后需要说明的一点是，性能的最佳实践和质量控制以及安全控制的做法类似。一个通用的最佳实践就是雇用性能方面的专业人员以及质量控制和安全控制方面的优秀人员。

就安全来说，在查找性能问题上，人工智能或者神经网络性能优化工具的效果会比测试或人工性能测试专家好很多。一款相似的软件可以运用在架构和设计上，该工具在编码前就可以根据性能优化规则和算法提供最佳的性能优化方案。

一般来说，在处理复杂问题，如安全漏洞和性能问题上，人工智能和神经网络是值得推荐的。这些主题与自主计算或者应用交叉，并且这些应用都倾向于监控并提高本机的性能和质量。

2.40 软件的国际标准

由于软件工程领域并没有公认的认证和许可，因此大家使用的国际标准大相径庭。而且，当使用国际标准时，无论开发的应用程序是有用的、中立的，还是有害的，我们都缺乏能够将该标准应用于软件开发的证据。一些适用于软件的国际标准是由国际标准化组织 ISO 建立的。影响软件应用的标准的例子有：

- ❑ ISO / IEC 10181 的安全框架
- ❑ ISO 17799 安全
- ❑ 《萨班斯 - 奥克斯利法案》
- ❑ ISO / IEC 25030 软件产品质量需求
- ❑ ISO / IEC 9126-1 软件工程产品质量
- ❑ IEEE 730-1998 软件质量保证计划
- ❑ IEEE 1061-1992 软件指标
- ❑ ISO 9000-9003 质量管理
- ❑ ISO 9001:2000 质量管理体系

对于功能规模估算，也有一些国际标准。截至 2008 年，在实际能够产生的改善方面，国际标准有效性的数据仍然比较稀缺。

军事和国防应用遵循军事标准而不是国际质量管理体系 ISO 标准。本书中也将涉及一些其他标准。

2.41 软件中的知识产权保护

显然，在软件知识产权保护方面的最佳实践，就是从专利或知识产权律师事务所寻求

法律建议。只有知识产权律师才了解版权、专利、商标、服务商标、商业秘密、保密协议以及非竞争性协议的利弊，并能提供适当的指导。笔者当然不是律师，在本节或这本书里所提到的也不能作为法律建议来看。

依据以上的法律建议，技术学科应该慎重考虑，如敏感信息加密、电脑防火墙和入侵防护、办公室的物理安全以及机密的军事软件，甚至电脑的隔离和使用防止微波的保护屏。微波可以用来收集和分析计算机正在执行的任务并提取机密数据。

许多应用软件包含专有信息和算法。一些防御软件和军事软件可能包含机密信息。专利侵权诉讼和知识产权盗窃诉讼的数量越来越多，并且还有继续增长的趋势。与过去相比，通过黑客或贿赂工作人员来盗窃软件数据的案件也在增多。

商业软件供应商也担心盗版和非法复制软件的问题。这个问题的解决方案包括注册、激活和在某些情况下实际监控软件在客户电脑上的运行情况。然而，这些解决方案只有部分可行，在许多发展中国家甚至在工业化国家，复制非法商业软件的情况也是很常见的。

一个常见的解决方案是对关键规范和代码段进行加密。然而，这个方法会给开发团队带来很多后续问题，因为人们需要理解未加密信息的含义。

未来，一个合理的解决方案将结合云计算，也就是应用程序可以存在网络服务器上，而不是个人电脑上。虽然这种方法可以保护软件本身，但它也并不是没有问题的，其可能会受制于黑客从无线网络的拦截，甚至服务器拒绝访问。

知识产权的保护需要专家提供法律建议，且需要专家提供物理安全和网络安全方面的建议，而这里只给出了一些一般的建议。

要注意确保办公室、笔记本电脑、家用电脑的安全，当然还有电子邮件的安全。旅行或出国参观时，电脑、笔记本电脑被盗，甚至笔记本电脑丢失都有可能发生。很多公司禁止员工携带电脑到海外。

除了计算机安全外，有时候可能还需要限制U盘、DVD、可读写CD光盘以及其他可移动媒介的使用。一些公司和政府机构禁止员工携带可移动磁盘进出办公室。

如果你的公司支持家庭办公或远程办公，那么你的专有信息可能会有风险。虽然大多数的员工可能是诚实的，但他们的家庭成员可能会为了娱乐而尝试黑客行为。而且，因为不能控制员工家庭的无线网络，其中一些无线网络可能没有任何安全保证。

对于拥有自主知识产权公司的员工，通常需要他们签订某种形式的就业协议和非竞争协议。这是一个非常复杂的领域，一些公司索要员工创新和发明的所有权，不管这些发明是否与工作有关，这样的协议通常会抑制创新。

外包协议也应该被作为知识产权保护的一部分去考虑。显然，外包供应商需要签保密协议。相比其他国家，这些协议在美国更容易执行，这也是一个需要考虑的因素。

如果知识产权是嵌入在软件中的，比较谨慎的做法是在其中列入可以识别的特殊代码模式，那么就可以阻止盗用或非法复制软件了。

如果公司裁员或者停业，应该提供处理知识产权的特殊法律建议。对于裁员，显然所有离职员工需要签署非竞争性协议。对于停业，依据破产规则，知识产权可能是一种资产，所以它仍然需要受到保护。

申请专利是保护知识产权的一个主要方法，在软件行业中这方面争议很大。一些人认为专利是保护知识产权的主要策略；另一些人认为专利仅仅是榨取巨额费用的手段。而且，将来在专利法上可能会有一些变化，使软件知识产权的获得变得更加困难。软件专利的问题是非常复杂的，细节部分已经超出了本书讨论的范围。

在软件中，出现了一种新的保护算法和商业规则，该方法从“圣经代码（Bible Code）”衍生，它基于等距字母间距（Equidistant Letter Spacing, ELS）。

对《Genesis》这本书的统计分析发现，从被等距离分开的字母可以拼出单词甚至短语。软件创造者可能对注解或实际的说明使用相同的途径，使用 ELS（等距字母间距）方法嵌入少量的代码去识别这个软件的开发者。等距字母拼出的单词或词组就可以作为盗窃的证据，如“Stop Thief”。当然，如果小偷插入自己的 ELS 代码就可能会适得其反。

2.42 防止病毒、间谍软件以及黑客

截至 2009 年，信息的价值都快接近黄金、铂金、石油或其他昂贵商品的价值了。事实上，随着全球经济衰退的蔓延，信息价值的上升速度会比自然产品价值（如金属或石油的价值）上升得更快。由于信息价值的增长，也引发了更多复杂类型的偷窃。在过去，黑客和病毒通常是个人的所为，有时是学生在做，甚至被高中生拿来当做寻找刺激的方法。

然而，在如今的社会，窃取有价值的信息已经渗透到了集团犯罪、恐怖组织，甚至敌对外国政府。不仅如此，而且拒绝服务访问和“搜索工具”可以借用电脑强大复杂的功能来关闭企业数据中心，以干扰政府的运作。随着全球经济的下滑，这种情况会变得更糟。

电脑被用来存储有价值的信息，如财务记录、医疗记录、专利、商业秘密、机密军事信息、客户名单、地址、电子邮件地址、电话号码以及社会保险号码等，其存储信息的总价值达到数万亿美元。在当今的社会中，几乎没有像信息这样如此昂贵又如此容易被盗窃的商品了。

偷窃不仅增加了对软件和财务数据的威胁，同时也可能会增加对投票和选举软件的威胁。任何计算机通过任何方式连接到外围世界都是危险的。即使计算机从物理上是隔离的，但它仍然可能存在一些由电磁辐射带来的风险。

虽然许多组织，如国土安全、国防部、联邦调查局、国家安全局、IBM、微软、谷歌、Symantec、McAfee、卡巴斯基、计算机协会以及许多其他组织都有相当称职的安全人员和可靠的安全工具，但是整个工作需要有一个中心组织负责监控安全威胁并分发数据，这是做好防御工作的最佳实践。分散的安全软件很难组织起来抵抗所有的威胁，并监控未来的威胁。

美国联邦调查局（FBI）启动了一个叫做 InfraGuard 的合作组织，该组织在过去试图贡献软件和计算机安全问题方面的数据。根据 InfraGuard 网站的说法，《财富》500 强公司中有大约 350 家都是其成员。在许多大城市中，这个组织还有隶属于联邦调查局的办事处，如波士顿、芝加哥以及旧金山等。然而，规模较小的公司在应对安全问题上就没有像大型企业那么有优势了。成为 InfraGuard 的会员将是一个很好的开始，当然也是一个最佳实践。

美国国土安全部 (DHS) 为了保证软件安全 (SwA) 也成立了一个联合政府和企业的团体。当前, 这个组织已经发布了一份软件安全报告书 (Software Security State of the Art Report, SOAR), 该报告中涵盖了预防、防御和恢复软件安全缺陷的诸多内容。参加这个组织, 并遵循 SOAR 的原则进行讨论也是一个值得学习的最佳实践。

正如这本书所提到的, 国土安全部正在筹建一个重要的新型安全研究设施, 该设施可能作为人民政府机构中的中央协调机构, 并且也将在某些方面协助企业。

由罗德岛的众议员 James Langevin 所做的政府安全报告也即将发表, 该报告涉及所有已知的问题, 并且说明得也很详细, 而这无疑也提供了一些额外的指导。

遗憾的是, 很多安全方面的文献都着力于处理开发和部署之后的安全威胁。但需要将架构、设计以及开发的安全作为一个基本原则。Ken Hamer-Hodges 写了一本与此相关的书, 《Authorization Oriented Architecture》, 其中谈到更多的基本课题。在这些课题之中, 自动化计算机安全的问题将从用户转移到系统本身, 而这是通过详细的边界管理做到的。这就是为什么这些课题附加了功能上的讨论。同时, 安全框架也是一个最佳实践, 如谷歌的安全存储框架可以避免被转向钓鱼网站。自从新的 E 编程语言被设计成为最安全的程序语言之后, 使用 E 语言也成了最佳实践。

在安全方面, 培训业务分析师、系统分析师和架构师都没有跟上恶意软件的发展脚步, 因为威胁正变得更多更严重, 所以弥补这些漏洞的速度需要更快。

我们可以将软件的安全威胁比作内科感染。防火墙之于电脑病毒感染的作用, 就像对付生物危害防护服, 但是软件防火墙也可能被攻破。

其他的防御, 如杀毒软件和反间谍应用程序, 就像抗生素一样, 它们不仅能阻止一些感染的蔓延, 还能扼杀一些现有的感染。然而, 就像与医学抗生素一样, 有些感染会有抗防御性且不能被彻底去除。随着时间的推移, 抗药性感染的发展将比消除感染发展得更迅速, 这就是为什么多态软件病毒 (Polymorphic Software Viruses) 是病毒的最佳选择。

对于软件安全防护来说, 最好的策略就是, 通过更好的架构、更好的设计、更安全的编程语言以及更好的边界控制来改变软件应用程序的 DNA, 并增加它们对感染的自然免疫。

解决安全问题的方式是考虑科学的根基, 并且依据最小权限原则 (Principle of Least Authority) 建立物理术语的边界控制, 每个子程序的调用都被当作受保护对象的一个类实例。子程序中应该没有全局的条目, 没有全局的名称空间, 没有像 C:/ 目录 / 文件或 URL <http://123.456.789/file> 一样的全局路径名。每个子程序应该做边界检查, 并且调用也应该是受保护的。所有程序的引用都是本地名称的动态绑定, 并且在运行时一直执行访问控制检查。使用安全的语言和方法 (例如目前的 E 语言和 Caja 语言)。Hamer-Hodges 草案中所提到的一些常规最佳实践包括:

- 经常更改密码 (已被目前的技术淘汰)。
- 不要手动地点击邮件 URL 类型的链接。
- 在所有的收件箱禁用预览窗口。
- 在文本格式下阅读电子邮件。
- 不要打开邮件的附件。

- 不要启用 Java、JS 或特别的 ActiveX。
- 在您的 Web 站点不要显示你的电子邮件地址。
- 不要在不知道链接地址的情况下点击链接。
- 不要让电脑保存你的密码。
- 不要信任邮件消息的来源。
- 将软件或系统升级到最新的安全水平，IE 浏览器尤其如此。
- 考虑改用 Firefox 或 Chrome。
- 只运行可信任的应用程序。
- 在下载的时候，阅读用户协议（它们可能会出卖您的个人数据）。
- 拒绝可能携带病毒和蠕虫的电子邮件。
- 不允许弹出窗口。
- 拒绝应用程序额外权限的要求。
- 拒绝在桌面以外地方执行编辑请求。
- 拒绝应用程序在其他方面编辑权限的要求。
- 拒绝用网络连接奇怪或不被信任的内容。
- 当安装应用程序时，应在新的文件夹路径上提供一个新名字、新图标。
- 拒绝访问特定网站之外的站点。
- 拒绝所有不受信任的访问以防被攻击。

截至 2009 年，网络安全依然处于非常危险的境地，对有经验的计算机用户来说，拥有两台电脑可能是最好的应对办法。其中一台用于网上冲浪和上网。另一台电脑不连接到互联网，并将只允许通过了病毒和间谍软件检查的媒介访问。

惊人的是，黑客们是有组织的。目前，有专门的期刊、网站以及课程来培训黑客。事实上，通过查阅文献，我们可以找到许多讲解如何破解和抵御黑客的信息。2009 年，黑客“行业”比安全行业似乎更大更复杂。考虑到信息价值的增长以及计算机安全的根本缺陷，这似乎并不足为奇。目前，似乎并没有某项统计是针对黑客或安全专家的，但截至 2009 年，黑客社区的增长速度可能超过了安全社区。

标准的最佳实践包括使用防火墙、防病毒软件、反间谍软件以及物理安全保障措施等。随着黑客和安全公司之间的角逐不断加剧，日常提高警惕是必需的。应该保持病毒库每日更新。例如，目前生物防御法也是一种最佳实践，该方法允许我们使用指纹或视网膜作为验证，以获得软件或电脑的使用权限。

截至 2009 年，身份盗窃保险和公司的网站认证（如 VeriSign）都是模棱两可的。至于身份盗窃保险，这个想法似乎是合理的，但我们需要的不仅仅是偿还损失和费用，而是更加积极的支持措施。其中一个最佳实践是，使公司或非营利组织直接与所有的信用卡公司、信贷部门以及警察部门联系，并且在身份被盗时，能够快速反应，及时给消费者提供帮助。

至于认证的网站，在线搜索这类主题几乎揭示了与利益一样多的问题和错误。在这方面，这个想法可能是有效的，但是实现起来就不那么简单了。

接下来，我们将讨论当前网络世界中的一些主要威胁：

广告软件

由于电脑的广泛使用,计算机已经成了一个主流的广告媒介。许多软件公司的收入来源就是在软件中植入广告,软件运行时广告就会显示。事实上,对于共享软件和免费软件来说,植入广告可能是主要的收入来源。例如,Eudora 电子邮件的客户端就是由广告来支持运营的。如果广告软件只是显示信息,那么这将会使人厌烦而不会带来什么危害。然而,广告软件不仅可以收集信息还可以显示这些信息。如果是这样的话,广告软件就超越了界限,成了间谍软件。截至 2009 年,一般的消费者是很难区分广告软件和间谍软件的。所以安装反间谍软件是最好的办法,虽然并不是完全有效。实际上,高级计算机可能会安装三个到四个不同的反间谍软件,因为没有哪一个会是完全有效的。

认证、授权以及访问

计算机和软件都有一个方法层次,该方法层次可以保证使用计算机或软件必须经过授权。对于普通用户,如果没有管理访问权限,许多功能都是无法使用的。在计算机或软件第一次安装时,就确定了其访问的权限。管理员会分配给其他用户各种权限和访问授权。用户需要取得管理员的同意或身份授权验证,才能取得计算机或软件的使用权限。不仅是用户,软件应用也需要取得证书的访问权限。虽然用户的身份验证不是微不足道的,但它也不会产生大量歧义。例如,视网膜扫描或指纹扫描就提供了准确识别用户的方法。然而,在安全链中的身份验证和软件授权似乎是一个薄弱环节。访问控制列表(Access Control Lists, ACL)是唯一可用的最佳实践,但是它只是针对静态文件、服务以及网络。访问控制列表不能够区分身份,所以当病毒或者木马有相同的授权时,就会被误视为终端用户。如果一些授权软件包含蠕虫、病毒或者其他形式的恶意软件,它们可能使用访问权限来繁殖。截至 2009 年,这个问题的复杂程度已经导致没有应对日常授权的最佳实践了。然而,有一种基于容量安全的特殊形式授权是理论上的最佳实践。可惜的是,基于容量安全的最佳实践是最复杂的,且未被广泛应用。历史上,Plessey 250 计算机按顺序安装了一个基于硬件的容量模型,以便阻止黑客攻击以及未经授权的访问列表变化。这种方法已经使用了很多年,并且随着谷歌的 Caja 以及 E 语言的流行而再度引起大家的注目。

后门程序

通常情况下,我们需要使用密码和登录程序来登录。后门程序是指,在不使用密码和用户名以及其他协议的情况下,绕过程序的正常入口而进入软件的程序。错误处理例程和缓冲区溢出是常见的后门入口。一些计算机蠕虫安装在后门程序中,它们可以被用来发送垃圾邮件或执行有害行为。一个令人惊讶的事实是,程序开发人员会有意地在应用程序中加入后门程序。这就是为什么机密软件以及涉及金融数据的软件需要进行仔细的审查、静态分析,并且要检查软件开发团队安全背景。令人担忧的是,如果编译器开发人员在程序中加入一个类似函数,那么后门程序也会被植入。后门程序是微妙且难以防御的。在静态分析软件中使用特殊的人工智能可能是一个很好的方法,但是也还会有一些复杂且难以处理的问题。目前,最好的方法有以下几个:(1)假设错误是执行时被攻击的迹象;(2)永远不用户使用编码恢复程序在特权级别运行;(3)不使用全局路径寻找 URL 和网络文件;(4)

本地命名空间应该由可信任设备翻译。

僵尸网络

“僵尸网络”这个术语指的是一组“软件机器人”，它们可以自动操控并试图控制一个网络上的所有计算机，然后把它们变成“僵尸电脑”。这些僵尸电脑被僵尸牧人控制之后，会被用来做很多坏事，比如拒绝服务攻击或发送垃圾邮件。事实上，这种做法已经非常普遍了，以至于僵尸牧人甚至会出售这种服务给垃圾邮件发送者。僵尸网络往往非常复杂，并且也很难防御。虽然防火墙和指纹能够结合使用，但该组合也并不是百分百奏效的。时刻保持警惕，并且使用顶级的安全专家都是最佳实践。一些安全公司会提供僵尸网络的保护措施，并且它们一般都是通过比较复杂的人工智能技术来实现的。值得警惕的是，网络犯罪与网络防御正处于激烈角逐中。缺乏边界控制是让僵尸网络横行霸道的原因。基础架构变化、Caja 的使用以及安全的语言（如 E 语言）都能够阻止僵尸网络。

浏览器劫持

这是一个十分恼人，并且威胁很大的问题，它的产生是由于软件覆盖了正常的浏览器地址并且将其引导到其他地址。浏览器劫持一般用于市场营销，有时会引向色情网站或其他不良网站。最近出现的一个浏览器劫持形式叫“流氓安全网站”。一个弹出式广告将显示一条消息，如“您的计算机被感染”，并引导用户到某个要钱的安全网站。现代的反间谍工具在很大程度上都能够阻止并清除这些浏览器劫持。而这也是解决这类问题的最佳实践，但是，它们必须经常更新定义。一些浏览器，如谷歌的 Chrome 和 Firefox 浏览器，会将流氓网站保存到列表中，并在合适的时候警告用户。所以说通过列表来保存流氓网站也是一个不错的实践。

cookie

cookie 就是从网站下载到用户电脑上的一些数据片段。一旦下载，它们就能够在用户和软件商之间来回传输。尽管 cookie 包含一些用户数据，但它们不是软件，而是一些无源数据。cookie 最有用的地方一般是一些购物网站，像亚马逊就会用其捕捉用户的使用偏好。有的甚至将 cookie 用于捕捉用户信息并将其用于不良的目的。近几年，美国中央情报局和美国国家安全局都能够下载到访问其网站的任何一台计算机的 cookie 数据，这样就能够列出访问它们网站的名单。当然，cookie 也能够被劫持或被黑客更改。在未授权情况下对 cookie 的更改叫做 cookie 中毒。这也可以被用来更改网络商店的销售数据。浏览器也可以启用或禁用 cookie。cookie 可以是有益的，也能被用作有害的方面，而一般情况下并没有有效方法对其加以控制。笔者个人的做法是，除非某个网站出于商务原因需要用到 cookie，否则最好禁用 cookie。

网络勒索

像银行记录、病历记录或者商业机密这类有价值的信息被窃取后，接下来会怎样呢？一个令人惊讶的新犯罪形式就是“网络勒索”，罪犯一般会使用各种威胁手段，如将信息出售给竞争对手或将信息公开，再转过来将信息卖给被害人。这种新式的犯罪主要是针对公司而不是个人。公司的数据越重要就越容易被人当作窃取的目标。在这个问题上，最佳的

应对方法就是时刻保持警惕，并使用一流的安保人员、防火墙、安全软件。另外，在遭受勒索后要及时告知当局，如联邦调查局或侦测网络犯罪的警队。

网络骚扰

像 YouTube、MySpace 以及 Facebook 这类社交网络的出现，使得数以百万计的人们能够在未谋面的情况下就彼此交流。这些网络也带来了新的威胁，如网络欺诈和网络威胁。使用搜索引擎和互联网很容易搜集个人信息，更容易的是通过社交网络来传播这类信息以编造谣言、诬告和损害个人声誉。因为网络跟踪能够匿名操作，尽管有些网络跟踪者会被逮捕和起诉，但是一般还是很难被跟踪。由于这个问题越来越普遍，许多国家，或联邦政府，正通过制定新的法律来禁止这种行为。网络跟踪包括联系警察或者其他当局，如果可以的话，联系跟踪者的网络服务商。如果各个社交网络里禁止使用匿名信息，可能就会减少或阻止这类犯罪，但也会让社交网络失去其存在的意义。

拒绝服务

这种形式的网络犯罪，是通过提供虚假信息 and 数据，来试图阻止特定的计算机、网络以及服务商的正常运营。这是一种非常复杂的攻击，它的实施需要相当的技巧和努力，所以在防御和阻止上也需要相当的技巧和努力。拒绝服务（DoS）攻击大约开始于 2001 年，当时攻击的是美国在线（AOL），并且安全人员花了一周的时间才使其停止。但是，后来又衍生出了许多类似的 DoS 攻击。这种攻击的前奏就是会向许多电脑发送蠕虫或搜索机器人，然后将它们都变成僵尸电脑，并使其在不知情的情况下也参与到攻击中去。这是一个复杂的问题，然而最佳的解决办法就是使用一流的安全专家来不断提高防御。

电磁脉冲

核爆炸的一个副产品就是电磁辐射的脉冲，它强大到足以能够摧毁晶体管以及其他的电气设备。事实上，这种脉冲几乎能够让 15 英里内的所有电气装置统统停止工作。这所带来的损失是非常严重的，以至于这些设备根本无法修理，比如电脑、音箱、手机等。电磁脉冲效应引发了电磁脉冲弹的研究。这是一种高空导弹，它能够在 50 英里内爆炸并引起电源关闭，并损坏周围不计其数的电子设备。不仅核爆炸能够产生脉冲，其他形式的轰炸也能够发出这类脉冲。虽然我们可以使用法拉第罩或将它们围在金属层里隔绝，但是这种做法对于一般民众来说是不实际的。像美国和俄罗斯这种军事国家早已开始积极研制电磁脉冲弹，并可能已经研制成功。很有可能的是，像朝鲜这类的国家也可能已经拥有这样的设备。电磁脉冲弹的出现对各国的经济发展都是极大的威胁，而且毫无疑问的是，很多恐怖组织也可能拥有这样的装置，但是普通民众是没有办法抵御它的。

电磁辐射

普通消费者使用家用电脑可能不用担心由于电磁辐射引起的数据丢失，但是对于军事和机密数据中心来说，这就是非常严重的问题了。计算机在运行中，会辐射出各种电磁能量，并且可以通过各种手段远程接收一些磁盘的数据和信息。早在 19 世纪 70 年代，人们就发现可以通过电磁辐射提取信息。通过电磁辐射获取信息要求有非常专业的设备和专业

的操作人员，还要有一般黑客获取不到的专业软件。电磁辐射在平民中的威胁也是存在的，例如在识别过程中，可能通过电磁辐射来破解“智能卡”信息。应对这个问题的最好方法就是将设备与铜或钢的物质进行物理隔离。当然，不断提高警惕性，并且聘请一流的安全专家也都是不错的方法。还有一个方法就是在数据中心安装信号强度高于计算机的电磁发电机，这样一来就可以干扰检测。这个方法类似于通过干扰来关闭海盗电台。

黑客

“黑客”这个词的出现要比计算机早很多，并且在很多领域都有不同的含义。然而，在本书中，黑客指蓄意攻入计算机或软件并修改其操作的行为的人。然而有一些黑客是恶意的并且是有害的，但是有些却是有益的。事实上，许多的安全公司和软件生产商都会雇用黑客来入侵软件和硬件以发现可修复的薄弱环节，然后加以巩固。虽然防火墙、反病毒、反间谍软件都可以达到这一目的，但是最好还是雇用一些好的黑客，让他们渗透到计算机和关键应用中以查找问题。

身份盗窃

盗用他人身份，可以用于消费、建立信用卡账户或从银行提款，这是人类历史上发展最快的一项犯罪行为。身份盗窃的最新用途是窃取医疗福利。事实上，窃取医生身份，甚至可以用在医疗保险制度和保险公司的欺诈索赔上。不幸的是，这类犯罪太多了，因为这仅仅需要一些现代的电脑技能和一般可用的信息就可以实现，如社会保险号码、出生日期、父母姓名以及其他的一些信息。值得警惕的是，许多身份窃贼就是受害者的亲戚或所谓的朋友。同时，身份信息也会被黑客用来出售或交易。基本上，每个电脑用户每天都会收到“钓鱼网站”的电子邮件，它们试图骗取你的账号和其他信息。随着全球经济的衰退，身份窃贼自然会有所增加。据笔者估计，至少有 50% 的美国公民的信息存在风险。避免身份信息被盗的最好办法包括经常检查信用卡、使用防病毒和反间谍软件等，还有就是确保信用卡、社会保险卡以及其他存有信息的卡片不要丢失。

击键记录

在软件行业刚兴起的时候，击键记录技术就已经成家用计算机最严重的威胁之一了。软件和硬件都有击键记录的方法，但是电脑用户更容易碰到软件击键记录。有趣的是，击键记录在研究用户性能方面也发挥着积极作用。在当今世界，不仅是按键记录，鼠标移动和屏幕触碰都需要技术记录。击键记录技术最恶意的使用方式就是用来截获密码和安全码，然后再窃取银行账户、医疗记录或者其他专有数据。不仅计算机有此危险，ATM 机也存在类似风险。事实上，这种技术也可以用于投票机，用来影响选举的结果。对此反间谍软件是比较有用的，还有就是一次性密码也可以有效避免类似问题。这是一个相当复杂的问题，因此目前的最佳实践就是深入研究并寻找解决问题的新办法。

恶意软件

这是一个由“恶意”和“软件”混合组成的术语。这是各种安全问题的通用描述，如病毒、间谍软件、木马以及蠕虫等。

钓鱼

这个词源于真正意义上的“钓鱼”，但是它是指试图诱导计算机用户泄露机密信息的做法。例如这个程序会发送虚假邮件，让人们觉得这些邮件是由银行或其他合法企业发出的，然后再骗取机密信息。一个“钓鱼”相关的典型案例是发生在尼日利亚的那起，一封政府发出的邮件声称其无法将资金转移出境，而希望将其存放在美国账户。这封邮件要求收件人回复他们的账户信息。这种早期的“钓鱼”行为很快就会被识破，除了极少数的人会上当，几乎没有人会上当。不幸的是，现代的“钓鱼”技术更加成熟且难以辨别。最好的办法就是，不要理睬任何要求回复个人信息或账户信息的邮件。但是，新的“钓鱼”形式会更加复杂，它可以拦截浏览器。例如，当你想要访问 eBay 或者 PayPal 时，浏览器会被重新定向到一个假的网站，但其看起来就像真的一样。不仅有导向虚假网站的，还有电话号码误导的。然而，随着“钓鱼”变得越来越复杂，检测也就越来越困难了。幸运的是，信用卡公司、银行和其他可能会被“钓鱼”的机构已经成立了一个反“钓鱼”的非营利组织。对于软件公司来说，加入这样的组织可以有效防御这种问题。而对于个人来说，验证电话号码并拒绝回复要求个人信息或账户信息的电子邮件都是很好的应对办法。像 Firefox 和 IE 这些浏览器都备有钓鱼网站和黑名单，因此当用户不小心浏览到黑名单中的网站时，浏览器就会向用户发出警告。编程语言 C++ 和 E 语言都能够很有效地抵御网络钓鱼。

物理安全

为了确保数据中心的物理安全，最好方法是将数据存储在笔记本电脑或拇指驱动器中，或者使用无线传输。在报纸和杂志上，几乎每个星期都可以看到一些机密数据丢失的新闻，这些一般是由于笔记本电脑丢失或笔记本电脑遭盗窃引发的。在确保数据的安全上，有很多有效的方法，但是我们得考虑到才行。目前，物理安全保护模式包括指纹识别和视网膜识别，它们作为访问电脑或应用程序的密码。

盗版

在当今社会，盗版可以说无处不在，并且还形式多样。海盗问题可能只会让非洲沿海国家担忧，但是软件盗版的扩张却是令世人担忧的。虽然日本和亚太地区是众所周知的盗版来源地，但是伊朗和美国的争端也使得伊朗无视知识产权的保护，在伊朗，人们可以无限制地复制软件。这就意味着中东地区也会是盗版软件的温床。在美国和其他一些拥有严格知识产权保护的国家和地区，微软和其他大型软件厂商正在积极地使用法律武器打击盗版。非营利商业软件联盟甚至会奖励举报盗版的人。未经授权复制软件确实是一个严重的问题。对于小型的软件供应商来说，一般的预防措施就是通过注册和激活才能使用其产品。有趣的是，在公开源码和免费软件社区中处理办法则完全不同。例如，公开源码的软件公司通常会使用静态分析方法去寻找一些安全漏洞。当然也会有十几名开发人员审查存在问题的代码，然后解决安全漏洞问题。

rootkit

在 UNIX 操作系统中，root 指的是某个拥有改变操作系统或内核权限的用户；对于 Windows 系统来说，管理员权限则是与之等价的。rootkit 是一个程序，它能够渗透到计算

机中并且掌握操作系统的控制权。一旦控制了系统的权限，Rootkit 就可以用来发动拒绝服务攻击、盗取信息、重新格式化磁盘或者执行一些恶作剧程序。2005 年，索尼公司故意在 CD 中安装一个 rootkit 来防止通过计算机复制盗版音乐。然而，这却带来了一个意外的结果，那就是黑客、间谍软件和病毒可以通过这个 rootkit 进入电脑。毋庸置疑的是，一旦索尼的这一做法被媒体曝光，公众的抗议会迫使索尼撤销这个 rootkit。rootkit 很微妙，有时不仅能够躲过一些杀毒软件，还能够攻击杀毒软件本身。截至 2009 年，尽管像诺顿和卡巴斯基这些安全公司已经研发出了一些专门寻找 rootkit 的办法来进行自我保护，但目前还是没有应对这一问题的好办法。

智能卡劫持

最近才刚刚发展起来的威胁形式是通过远程读取各种“智能卡”进行的。这些卡片里包含了个人信息。智能卡包括一些嵌入信息的新型信用卡和护照。政府正在积极敦促市民将这些卡片放置在金属套或金属箱中，因为这些卡片上的数据能够在 10 英尺范围内被获取。顺便说一句，乘客通常所使用的“EZ 通”卡也存在风险。

垃圾邮件

垃圾邮件的英文“Spam”最原始的意义是指肉类产品，但其网络定义是指那些垃圾广告、垃圾邮件或者即时消息中附加的广告。现在互联网是世界上最主要的沟通媒介，全球大约有五分之一的人会使用网络，所以利用互联网进行市场营销和广告投放是大势所趋。垃圾邮件数量十分惊人，在所有的电子邮件中，大约有 85% 都是垃圾邮件，这显然会降低互联网以及其他一些服务的速度。垃圾邮件很难防御，这是因为有些邮件是来自僵尸电脑的，这些电脑被蠕虫或其他病毒劫持，然后就不知不觉成为了传送垃圾邮件的帮凶。一些国家已经通过立法来阻止垃圾邮件，但是发送垃圾邮件很容易外包到其他没有相关立法的地区进行。与垃圾邮件一起兴起的还有一个行业，那就是电子邮件地址获取。搜索机器人可以搜索电子邮件地址，一旦发现或收集到这类信息，就可以将其作为商品出售。另一种途径是通过社交网络来收集电邮地址。他们的服务协议中明确指出用户邮箱会被公开并有可能被出售。应对垃圾邮件的一个最佳途径，就是使用间谍软件和垃圾邮件拦截器，但它们都不是完全有效的。一些垃圾信息网络可能会被隔绝，禁止它与其他网络联系，但这在技术上是具有难度的并且有可能会遭到起诉。

鱼叉式网络钓鱼

这个词指的是新型且非常复杂的网络钓鱼，为了欺骗受害者，它们会在其发送的钓鱼电子邮件中包含大量的个人信息。它和普通网络钓鱼的区别就是这种形式中包含个人信息。例如，一封表面上来自朋友或同事的邮件肯定会比来源不明的邮件更加让你信任。因此，鱼叉式网络钓鱼是非常难以防御的。通常黑客会入侵公司电脑并发送鱼叉式钓鱼邮件给公司所有的员工，并让大家认为这封邮件是由会计、人力资源或其他组织发出。实际上，经理真正的姓名也可能包含在里面。应对鱼叉式网络钓鱼的最好办法，就是避免将个人或财务信息回复给任何邮件。如果这封邮件看起来是合法的，那么也要在回复前进行电话复核。然而，鱼叉式网络钓鱼不仅只有网络诈骗，它还包含虚假短信和电话。

间谍软件

间谍软件是指那些安装在主机上，能够控制操作系统和浏览器的软件。间谍软件的主要用途是显示垃圾广告，将浏览器重新定向到特定的网站并提取个人信息、盗窃身份。在微软的 IE 7.0 浏览器出来之前，几乎没有可以下载并执行 ActiveX 控件的浏览器。这很快就被黑客发现，他们可以劫持计算机浏览器并投放广告。因为间谍软件经常嵌入在注册表中，所以很难去除。截至 2009 年，防火墙和现代反间谍软件的结合使用能够抵御大部分间谍软件的入侵，并且能够消除大多数间谍软件。然而，在黑客和保护者的激烈技术竞争中，黑客们往往会领先一步。尽管与使用微软系统的电脑相比，Macintosh 电脑更不容易被间谍软件入侵，但是如今不受间谍软件影响的电脑已经不存在了。

特洛伊木马

这个术语源于著名的故事“特洛伊木马”。在软件的语境下，木马是指，那些看似有用但是在欺骗用户下载或通过磁盘安装后，便开始控制计算机或者非法获取个人信息的某种软件。木马相关的一个典型形式是包含木马的屏保。一些景色美丽的屏幕保护是提供免费下载的，如瀑布或者是环礁湖。然而，在这里面却隐藏着会给计算机带来危害的恶意软件程序。木马程序通常会涉及拒绝服务攻击、身份盗窃、击键记录以及很多其他有害的行为。目前的杀毒软件通常能够有效预防木马，所以应对这一问题的最好办法就是安装、运行并及时更新这些杀毒软件。

计算机病毒

计算机病毒起源于 20 世纪 70 年代，并在 20 世纪 80 年代成为一个棘手的问题。计算机病毒就像疾病病毒一样，会侵入一个宿主并在其中大量繁殖，然后离开原来的宿主并寻找新的宿主。计算机病毒仅仅通过复制和传播就能够降低网络运行的速度并导致性能降低。此外，一些病毒也有蓄意损坏电脑、窃取私人信息以及其他一些恶意行为。例如，病毒会盗取地址然后发送带有病毒的软件给每个联系人和原来的主机。宏病毒通过微软的 Word 或 Excel 创建文件来传播病毒，这种做法很普遍却也很难处理。通过即时信息传播的病毒处理起来也很麻烦。病毒通常附着在文档、邮件或即时信息中来传播。由于病毒开发人员很聪明且精力充沛，所以使用杀毒软件仍然是最有效的应对方法。一些新型病毒的变体能够通过改变自己来避开杀毒软件。这些变异的病毒叫做多态病毒。尽管病毒主要攻击微软系统，其他的操作系统也存在危险，包括 Linux、UNIX、Mac 以及塞班操作系统等。安装杀毒软件并不断更新病毒库是应对病毒问题的最好办法。经常使用检查点和恢复点也是个办法。

网络捕鲸

网络捕鲸也是一种网络钓鱼，但是它的目标是公司的高管、总裁、高级副总裁、首席执行官、首席信息官以及董事会成员等。网络捕鲸往往是非常复杂的。有个例子是，一封邮件自称其来自著名律师事务所，邮件中说明了诉讼目标或他（或她）的公司。有时或者可能是一封“谁是谁”的邮件请求，或者伪装成著名的商业杂志发出的请求。应对这类问题的最好方法是，在没有通过电话或其他方法查明的情况下，拒绝回复邮件。

无线安全泄露

现代世界，使用无线计算机网络就像使用手机一样普遍了。许多家庭像公共场所一样，都有无线网络，还有一些城市提供无线网络覆盖。随着无线网络成为商务之间以及人们之间交流的标准方法，无线网络也吸引了很多黑客、身份盗贼以及其他形式的网络犯罪人员的注意。未受保护的无线网络允许网络犯罪者访问并控制计算机、重新定向浏览器并盗取私人信息。其他一些并不明显的活动也是有害的，例如，未受保护的无线网络可被用于访问色情网站或在不被发觉的情况下向第三方发送恶意电子邮件。由于很多顾客和计算机用户对计算机和无线网络都不精通，大约有 75% 的家庭计算机网络未受保护。一些黑客甚至穿行于各大城市中搜寻未经保护的网路（这也叫做“沿街扫描（war driving）”）进行犯罪。事实上，在人行道和建筑上还有特殊的标志和符号来表示未受保护的网路。在许多咖啡店和旅馆内的网路都未经保护。应对这一问题的最佳方法就是，使用最新的密码和保护工具，加密或通过经常更改密码的方法来避免出现无线网络安全漏洞。

蠕虫

能够自我复制，然后通过网络在计算机之间传播的小型软件应用程序叫做蠕虫。蠕虫和病毒很相似，但是它是自动传播的而不是通过电子邮件或文档传播的。而且有一些蠕虫是良性的（微软曾经试图在操作系统补丁里安装蠕虫），但是多数是有害的。如果一些蠕虫成功地复制并在一个网路中传播，那么它们就会使用大量的带宽并且降低网路的性能。更糟糕的是，有一些蠕虫还拥有子例程或载荷，它们能够执行有害或恶意的任务，如清理文件。蠕虫还能被用来创建僵尸电脑或参与到拒绝服务攻击中。为了避免蠕虫，最佳的办法包括从操作系统供应商（如微软系统）处下载安装最新的安全更新，或者使用杀毒软件（要经常更新病毒库）和防火墙。

从现代计算机和软件的这些危害中，我们可以看出，要保护计算机和软件免受危害，我们还需要从长计议，并不断地提高警惕。安装和使用各种保护软件也是很有必要的。最后我们还要考虑到电脑安全、亲戚朋友在使用我们电脑时会产生问题。安全问题比全球经济衰退都要普遍。信息是一种商品，无论经济领域发生了什么，信息的价值都会增长。此外犯罪组织和主要的恐怖组织都是黑客，他们是拒绝服务攻击和其他网络犯罪的主要制造者。

如果你打破了软件安全的经济学，那么分配的成本就会远不如 2009 年。从数据来看，在公司安全成本中，大约有 60% 是用于数据中心的防卫与安装软件，大概 35% 是用在回复攻击和拒绝服务上，只有 5% 是用于建立预防措施。假设一个《财富》500 强公司的年度安全成本是 5000 万美元，则故障预防可能是 3000 万美元，1750 万美元用于恢复，而只有 250 万美元是用于开发预防程序的。

在架构、设计、安全编码实践、边界控制以及语言（如 E 语言）上都用更有效的预防方法，那么未来的成本分布应该是 60% 用于预防，35% 用于恢复，5% 用于防御。预防措施提高后，总成本也会相应地降低：或许总成本就可以由每年 5000 万美元变为 2500 万美元。预防成本将为 1500 万美元，防御成本将是 875 万美元，恢复成本将只有 125 万美元。表 2-9 显示了两个成本资料。

表 2-9 软件安全成本预估 (2009 ~ 2019 年, 世界 500 强公司)

	2009 年	2019 年	差
预防	2 500 000 美元	15 000 000 美元	12 500 000 美元
防御	30 000 000 美元	8 750 000 美元	-21 250 000 美元
恢复	17 500 000 美元	1 250 000 美元	-16 250 000 美元
总计	50 000 000 美元	25 000 000 美元	-25 000 000 美元

因此, 只要软件的安全性在很大程度上取决于人们明智地更新病毒的定义, 并且安装反间谍软件, 那么我们就不可能在安全上取得重大的突破。软件产业最需要的是, 设计并开发出防御性更强的软件和操作系统, 然后让其完全自动地防御, 并且几乎不需要人工参与。

2.43 软件的部署和定制

很少会有软件文献涉及开发和维护之间的问题, 这是一个灰色地带, 如软件应用程序资源的部署和安装。考虑到大型软件包的部署和安装, 如企业资源规划 (ERP) 工具要用时超过 12 个月, 花费超过 100 万, 涉及超过 25 个顾问和 30 个内部人员, 资源部署这个问题需要更多的研究以及更多的文献资料。

对于大多数使用个人电脑或苹果电脑的人, 软件的安装和部署是通过网络或从 CD 或 DVD 进行的。虽然一些软件安装太麻烦 (比如 Symantec 或微软的 Vista), 但多数都可以在几分钟内完成。

遗憾的是, 对于大型应用程序来说, 就不仅仅是装好然后开始工作的问题了。为了与现有的应用兼容, 大型应用程序 (如 ERP) 一般需要大量的定制。

此外, 新版本通常会有很多漏洞, 所以经常更新和维修也是安装过程的一部分。

同时, 拥有成百上千用户的大型应用需要培训不同类型的用户。虽然供应商可能提供一些培训, 但他们不知道客户购买软件后的具体用途。所以通常情况下, 企业必须把很多的定制课程放在一起。所幸有些工具和软件包可以帮助大型应用程序进行内部培训。

因缺陷和学习问题的存在, 仅停止使用旧应用程序并同时使用新的商业软件是不可行的。通常情况下, 在新应用最初运行的几个月后, 主要的任务是检查错误, 并且提高用户的使用速度。

简而言之, 一项重要新软件包的资源部署可以运行 6 到 12 个月之多, 并且其中还会涉及很多类型的人员, 如顾问、培训师以及需要学习软件的内部人员等。部署的最佳实践范例包括:

- ☐ 如果有的话, 可以加入新应用的用户协会。
- ☐ 采访正在使用软件的客户, 并获得一些软件部署相关的建议。
- ☐ 在部署方面寻找有工作经验的咨询师。
- ☐ 针对软件创建定制的课程。
- ☐ 针对新的应用程序的培训课程。
- ☐ 定制新的应用程序以满足当地需求。

- 开发新应用和旧应用之间的接口。
- 记录并报告在部署过程中遇到的错误或缺陷。
- 从供应商那里安装补丁和新版本。
- 评估新应用成功的可能性。

安装和部署大型软件是我们经常遇见的，但软件文献中相关的研究和报道却非常缺乏。任何活动的用时可以超过一年，花费可能超过 100 万美元，并且需要超过 50 个全职员工进行认真的分析工作。

资源部署的成本和危险似乎与应用程序的规模和类型有直接的关系。对于个人计算机软件 and 苹果电脑软件，资源部署通常是相当简单的，可以由客户自行执行。然而，一些公司，如 Symantec 不允许删除应用程序的旧版本，但正常的 Windows 删除的遗留痕迹会干扰新程序的安装。

大的应用程序，比如大型主机操作系统、ERP 软件包以及定制软件的安装都是非常麻烦，并且资源部署的成本也是相当昂贵的。此外，这类应用程序在被应用之前通常要求广泛的定制以适应当地的情况。当然，这种情况还需要对用户进行培训。

2.44 培训软件应用的客户或用户

在软件产业中有一个有趣的现象，那就是商业供应商提供培训和辅导信息，主要出版相关的指导性书籍，如 Vista、Quicken、微软的 Office 以及其他的几十种流行软件。同时，培训公司提供数十个软件包交互培训信息 CD。如本书所述，要想学习使用流行软件，就应该使用第三方的指导书籍，而不是供应商自己提供的。

对于 Oracle 和 SAP 发布的大型机专业应用程序，其他公司也提供有用户和维护人员的补充训练，通常也会比厂商自己做得更好。

在软件行业发展的 60 年中，大家可能认为标准用户培训资料应该有共同的特征，但实际上并不是这样的。

我们需要的是一系列的学习材料，包括但不限于如下这些：

- 特性和功能的概述
- 安装和启动
- 常见任务的基本用法
- 复杂任务的用法
- 按照主题分类的帮助信息
- 在出现问题时，故障排除的方法
- 常见问题解答 (Frequently Asked Questions, FAQ)
- 操作信息
- 维护信息

几年前，IBM 完成了一份用户评价相关的统计分析，所有软件说明书同 IBM 的软件一起提供给客户。那么评价最好的用户手册会被分发到所有的 IBM 技术文档撰写人员手中，

并建议他们以此为参考撰写新的用户手册。

当前,这种方法将有可能用于对第三方书籍做类似的研究,通过对亚马逊的技术书籍目录中所列的用户评论进行统计分析,那么各类书籍中最畅销的书籍可以作为新书的模式。

因为屏幕截图是静态且难以修改的,所以教程材料可能会迁移到网上,如电子书籍,可在亚马逊的 Kindle、索尼的 PR-505 等终端阅读。

虽然这些在 2009 年仍然是可望而不可及的,但在不久的将来,可能会有虚拟环境、虚拟形象以及三维模拟等更加复杂的在线培训。软件供应商的底线是他们提供的教程材料不足以培训客户。幸运的是,许多商业图书出版商和教育企业已经注意到了这些,并提供了更好的选择,至少有更广的用途。

以上的供应商和商业图书出版商、用户协会以及各种网站都有很多的志愿者,他们可以回答软件应用的相关问题。未来有可能会通过电子书来提供用户的信息,如亚马逊的 Kindle 以及索尼的 PR-505 等。

2.45 软件应用部署后的客户支持

软件应用程序的客户支持几乎没有令人满意的。除苹果、联想、IBM 等公司有相当好的客户评论外,其他数百家公司都受到批评,大多数是因为用户等待的时间太长或提供的支持信息客户不满意。

客户支持是劳动密集型工作,并且成本昂贵,这也是为何这方面很难做好的原因。在一个软件应用程序中,平均每 10 000 个功能点就需要一个客服人员。大约每 150 个客户需要一个客服人员。然而,随着使用量的上升,公司将无法承受越来越大的客服团队,所以客户与客服人员之间的比率最终会突破 1000 : 1,那么这意味着客户需要等待的时间会更长。因此,拥有 100 000 个功能点的大型软件假如有 100 000 个客户的话,那么需要的支援人员的数量将是非常巨大的,并且人数太少的话将会导致客户服务很难正常进行。

由于客户服务支持需要高成本和高劳动强度,所以通常的做法就是将这样的工作外包给一些像印度这样劳动成本较低的国家。

令人惊讶的是,只有几百个客户的小公司经常比大公司拥有更好的客服,因为他们的服务团队有很好的抗压能力。

提高客户服务质量的一个短期策略是提高软件的质量,以便减少软件交付时的漏洞。然而,并不是每一个公司都足够了解如何做到这一点。结合审查、静态分析以及测试可以提高缺陷去除效率,使其水平从今天的平均不到 85% 达到约 97%。减少错误或缺陷的软件会显著减少客户服务请求。

据笔者估计,在软件交付时减少 220 个左右的缺陷就可以减少一个客服人员。这个假设是基于客服人员每天回答大约 30 个电话,那每一个缺陷会被 30 个客户发现。换句话说,一个缺陷就会占用一个客服人员一天的时间,而每年有 220 个工作日。

一个更全面的长期策略将涉及许多不同的方法,包括一些新颖的和创新的方法:

❑ 开发具有人工智能的虚拟技术支持系统,让它们充当电话支持服务的第一梯队。因

为人员工资成本高且缺乏必要培训，所以虚拟服务系统可以做得更好。当然，这些虚拟的客服系统需要储备尽可能多的最新错误报告、临时解决方案以及主要问题等信息。

- 在客户和服务组织之间允许有简单的电子邮件联系。对于小公司或小的应用程序，这些可以由现场服务人员帮助完成。对于较大的应用程序或那些拥有数百万客户的程序，人工智能工具将扫描电子邮件，提供解决方案或者将问题传输到真正的服务人员那里进行处理。
- 将帮助信息和用户指南标准化，这样所有的软件应用都可以提供类似的信息给用户。这将加快用户的学习并允许用户在基本不损坏软件包的情况下，对软件包进行一些更改。这样做或许会促使国际标准化组织 (ISO)、IEEE 以及其他标准机构制定新的标准。
- 可重用的功能和特性，如提供可重用帮助、教程信息以及可重用的源代码。随着软件从定制开发到装配标准组件的转变，这些标准组件的教程材料必须是可重用构件的一部分，并且在许多软件应用中共享。

2.46 软件担保和召回

几乎所有的商业产品如果出现问题，在一段时间内都会有保修或包换的承诺，例如家用电器、汽车、照相机、电脑以及镜头等。软件产品却是一个例外。大多数的“软件质保”都明确说明不承保产品的适用性、质量以及对消费者造成的危害。大多数的软件产品都“明示或暗示”不承担质量保证。

一些供应商可能会为软件提供定期的更新或补丁修补，但是万一软件无法操作或操作失误的话，通常商家只会提供另一个副本，而这一份也会存在同样的缺陷。通常，软件不会退回供应商，而且也不会被退款，所以就更不用说是修复任何由软件给计算机带来的损害了，如损坏电脑文件或留下不可清除的痕迹。

所谓的软件质保，其实是一种最终用户许可协议 (EULA)，这是指用户在安装软件应用前就需要签署并确认。最终用户许可协议纯粹就是单方面的，并且主要是为了保护供应商利益而设计的。其原因就是薄弱的软件质量控制，这也是软件行业 50 多年来的最主要的弱点。

在本书撰写时，联邦政府正在努力起草一份统一计算机信息交易的法案 (UCITA)，该法案会作为统一商业代码的一部分。就目前来说，这一法案还存在很多争议，有的人认为在保护消费者方面，这甚至不如 EULA (最终用户许可协议)。如果事实真是这样，并且由于州立政府有权修改部分条款，这一法案的条例是很难统一的。

如果软件开发人员引入了实现缺陷去除效率达到 95% 以上的最佳实践，并且使用认证的可重用组件来构建软件，那么这将有可能创造一个能保证双方利益的公平质保条款。这样的条款应该包含如下几点：

- 如果客户不满意，供应商应该设定一个固定时间，如 30 天，将购买价格的全款退还顾客。

- 软件供应商应该保证软件能够按照用户指南中所提供的信息运行。
- 供应商应该提供免费的更新以及补丁修复，期限最少应该是购买后的 12 个月。
- 供应商应该保证，交付软件的物理媒介（如 CD 或 DVD 磁盘）中没有病毒或恶意软件。

除了以上这些具体的质保条款外，也应该包含以下几个主题：

- 软件帮助以及所有用户指南中都应该包含报告漏洞和缺陷的方法。
- 客户通过电话获取支持的等待时间不应该超过 3 分钟。
- 客户通过电子邮件获取支持的等待时间不应该超过 24 小时。

截至 2009 年，大部分的最终用户许可协议和软件质保条款都不尽如人意，这一点也是软件行业最值得改进的地方。

2.47 软件发布后的变更管理

理论上，软件交付后变更管理与软件交付前变更管理是一样的。这是因为，规格要根据需要不断更新，配置控制也要继续进行，而且用户报告的漏洞也应该包含在整体的漏洞数据库中。

实际上，软件交付后的变更管理通常没有软件交付前的变更管理严格。然而，对于代码的配置控制可能会继续，规格很少会保持不变。另外，一些小的漏洞修复和小的改进可能就没有做文档记录了。所以说，在使用 5 年后，应用程序的规格说明书就不再完整了。

同时，当产生了不能执行的“死代码”时，就需要对代码做更改了。代码注释也许已经过时。通过使用循环复杂度和基本复杂度来度量可能会使复杂性增加，所以变更常常会变得越来越困难。这种情况很常见，从而很多公司主要依靠长期任职的维护人员来负责更新，因为他们对于老化遗留代码的知识结构是成功更新的关键。

然而，遗留软件系统确实有一些强大的工具，它们可以帮助推出新版本，甚至能够有助于开发替代产品。因为在大多数情况下，源代码确实是存在的。所以可以在源代码中使用自动化方法，以提取隐藏的业务规则和算法，并将其用于推动应用程序替换或遗留程序的改造。这类工具的例子包含（但不限于）：

- 能够通过代码说明所有路径和分支的复杂性分析工具。
- 在选定语言中能够查找遗留代码漏洞的静态分析工具。
- 能够识别可通过手动移除的易错模块的静态分析工具。
- 能够识别可删除或可隔离的死代码的静态分析工具。
- 能够从代码中提取算法和业务规则的数据挖掘工具。
- 可以将传统语言转换成 Java 或现代语言的代码转换工具。
- 可以计算出遗留应用规模的功能点计算工具。
- 可以协助管理现存软件变更的翻新工作平台。
- 能够在代码段检查完后创建新测试用例的自动测试工具。
- 可以从当前测试用例库中显示差距和遗漏的测试覆盖工具。

除了自动化工具之外，遗留应用中源代码、测试库以及其他构件的正式审查也是非常有用的，当然构件还应时时更新。

随着全球经济陷入严重衰退，多年来一直运行的遗留应用可能会有重要的经济价值。然而，对结构简陋的遗留应用进行维护和改进，这样做是不符合成本效益的。对于遗留程序的结构与特征进行全面彻底的分析是非常必要的。因为人工处理可能无效且昂贵，像静态分析和数据挖掘这类自动工具，在未来的经济衰退周期里应该会非常有价值。

2.48 软件的维护和功能增强

由于“维护”这个词包含了很多不同类型的活动，因此在分析上，软件维护比软件开发更加复杂，并且也更加困难。同时，评估软件“维护”和“功能增强”，不仅需要评估变更本身，还需要详细完整地分析正在修改中的遗留应用的代码和结构。

截至 2009 年，已经有 23 种不同形式的工作被归入了“维护”的概念下。

通用术语“维护”的主要工作有：

1. 主要功能增强（超过 50 个功能点的新特性）。
2. 小规模功能增强（小于 5 个功能点的新特性）。
3. 维护（有效的缺陷修复）。
4. 保修期内的修复（正规合同下的缺陷维修）。
5. 客户支持（回复客户咨询或问题报告）。
6. 删除易错模块（删除复杂的代码段）。
7. 强制变更（必需的以及法定变更）。
8. 复杂性或结构分析（绘制控制流并加上复杂性度量）。
9. 代码重构（减小循环复杂度和基本复杂度）。
10. 优化（提高性能或吞吐量）。
11. 迁移（将软件从一个平台移至另一个）。
12. 转换（改变接口或文件结构）。
13. 逆向工程（从代码中提取潜在的设计信息）。
14. 再造工程（将遗留应用转化成现代的形式）。
15. 死代码移除（删除部分不能使用的）。
16. 删除休眠程序（归档未使用的软件）。
17. 本地化（修改软件以适应国际应用）。
18. 大规模更新，如欧元及 2000 年修复。
19. 重构或重新编程，以提高应用程序清晰度。
20. 退役（停止使用活动的服务 / 应用）。
21. 现场服务（向客户派遣维护人员）。
22. 向软件供应商报告漏洞或缺陷。
23. 安装供应商发布的更新。

尽管以上 23 个维护问题在很多方面都很不相同，但是它们都有一个共同的特点，那就是它们适合小组展开讨论：它们都是修改现存应用而不是重新开发一个新应用。

这 23 个修改形式中的每一个都是由不同的原因所导致的。然而，通常有很多可以同时进行。例如，在一个不断演变的版本中，同时执行功能增强和缺陷维修是非常普遍的。这些功能增强活动通常会有一般的序列或模式。例如，逆向工程往往先于再造工程，而且这两者经常一起构成联系体。对于大型应用和主要系统的版本，笔者通过对 6 到 10 种改进进行观察，发现这些都是在相同的版本上执行的。

近年来，信息技术基础架构库（Information Technology Infrastructure Library, ITIL）已经开始关注许多与维护相关的关键问题了，例如变更管理、可靠性、可用性以及很多用户日常遇到的重要问题。

由于随着时间的推移，老化软件程序的复杂性也持续增加，因此必须经常进行某种形式的改造或重构。截至 2009 年，一套处理遗留应用的最佳实践组合包含：

- ☐ 使用专业维护人员而不是开发人员。
- ☐ 考虑将维护外包给专业维护公司。
- ☐ 使用维护改进平台。
- ☐ 使用正式变更管理程序。
- ☐ 使用正式变更管理工具。
- ☐ 使用正式回归测试库。
- ☐ 对遗留应用执行自动化复杂性分析。
- ☐ 在遗留应用中寻找和消除所有易错模块。
- ☐ 识别遗留程序中所有死代码。
- ☐ 在主要维护前更新或重构应用程序。
- ☐ 在主要更新中使用正规设计和代码审查。
- ☐ 跟踪所有的顾客缺陷报告。
- ☐ 跟踪从问题出现到解决的响应时间。
- ☐ 跟踪从提交到完成的响应时间。
- ☐ 跟踪所有维修活动和成本。
- ☐ 跟踪商业软件的保修成本。
- ☐ 跟踪交付给客户的可用性软件。

目前，由于老化软件的维护和功能增强成本占据了整个软件行业主要的支出，所以使用先进的方法和工具来处理遗留应用是非常重要的。

在产品交付前，改进产品的质量能够减少后期的维护成本。因为负责维护的程序员每月大概能够修复 10 个漏洞，如果每个交付产品可以减少 120 个漏洞，那么就可以减少一个维护人员。因此缺陷预防、审查、静态分析以及更优的审查方法组合在一起就可以大大降低维护成本。在 2009 年，世界经济面临严重衰退之时，考虑这方面的因素显得至关重要。

截至 2009 年，一些较新的途径包括维护或创新工作台，这些工具由 Relativity Technologies 公司提供。这个工作台也具有一个新的功能，它能够快速准确地执行功能点分

析。在主要的改进前执行改造应该是一个常规项目。

由于许多遗留应用中易错模块都是非常复杂的，而且缺陷报告也是不成比例的，因此在进行重大变更时必须采取正确的方法。一般来说，在大型系统里面，约有 50% 以上的缺陷报告来自 5% 的模块。通常这些模块也是不可修复的，所以一旦发现问题，就只能通过手动移除或替代来解决这些问题。

截至 2009 年，维护外包已经成为软件外包中最受欢迎的形式。一般来说，软件维护外包协议比软件开发外包协议更容易成功，而且更少会有失败的案例或诉讼。一部分原因是维护外包公司的复杂性，另一部分原因是现在的软件公司不倾向于将风险巨大的大型软件开发项目外包给别的公司。

软件维护和开发都有一个共同的需求，那就是使用好的项目管理办法、高效的评估方法以及严格的生产率及质量评估。然而有 5% 的软件开发外包都以诉讼告终，相比之下软件维护外包在经济上更有优势和吸引力。

2.49 软件应用的更新和发布

一旦软件应用程序安装并使用，将会发生以下三件事：（1）会发现一些必须修复的缺陷；（2）根据法律规定和业务需求的变化加入新功能；（3）软件供应商通过推出新版本或增加新特性来赚钱。这部分的软件工程没有很完善的文献记录。在软件应用发布后，会涌现出许多对消费者或客户不利的做法，这些做法包括：

- ❑ 电话客户支持中顾客等待的时间过长。
- ❑ 电话支持对听力失聪的客户来说无济于事。
- ❑ 没有电子邮件客服或客户服务非常有限（如微软）。
- ❑ 客服人员收到问题无法解决。
- ❑ 客户服务向顾客收取费用，甚至报告问题也收费。
- ❑ 缺乏向供应商报告漏洞的方法（如微软）。
- ❑ 对报告的漏洞没有及时响应。
- ❑ 对报告的漏洞修复不到位。
- ❑ 过早地停止对旧版本软件的客户支持。
- ❑ 迫使顾客购买新版本。
- ❑ 强制性更改新版本文件格式。
- ❑ 不允许客户继续使用旧版本。
- ❑ 只保修能够替换的媒介，如磁盘。
- ❑ 只利于供应商的单方面协议。
- ❑ 新版本不能在旧版本基础上安装。
- ❑ 新版本中删除了有用的功能。
- ❑ 新版本与竞争软件不兼容。

以上这些做法非常普遍，现在很难找到一家公司在客户支持服务以及新版本发布方面

都做得很好的，尽管有一些不错的。所以截至 2009 年，以下是一些理论上的最佳实践：

- 最理想的情况是，在软件供应商的网站上应该展示出应用程序存在的漏洞和问题。
- 漏洞报告和帮助请求都应该可通过电子邮件进行传送，并且应该在 48 小时内回复。
- 电话客户服务应该在五分钟内接通。
- 当使用电话客户服务时，在客户被转接前，应该处理好 60% 的问题。
- 应该使那些听力障碍的顾客也能得到应有的客户支持服务。
- 收费的客户支持应该排除供应商已经公布的漏洞和缺陷。
- 当产品交付后，漏洞修复应该是能够自动安装的。
- 新版本和新特性的安装不应该要求手动卸载旧版本。
- 当文件格式改变后，供应商应该免费提供从旧格式到新格式的转换。
- 对于拥有上百万用户的应用，不能强制撤除客户支持服务。
- 不能迫使用户购买新版本，除非客户自愿获取新特性。

总的来说，大型主机供应商所提供的客户支持肯定比那些大批量生产个人电脑和 Macintosh 电脑软件的供应商好很多。然而，有些问题一直都是软件行业所特有的，如客户支持做得不到位、修补漏洞缓慢以及强迫用户安装新版本等。

2.50 遗留应用的终止或撤销

大型的软件应用一般能够使用很久。截至 2009 年，一些大型软件系统，如美国航空交通控制系统都已经连续使用了 30 年。许多大公司的主要大型内部应用程序都使用了 20 多年。

商业应用往往比一般的信息系统或系统软件的使用期更短，因为供应商会在产品售出一段时间后就会推出新版本并停止对旧版本的支持。像微软、Intuit 以及 Symantec，即使这些公司拥有数以千万计的用户且旧版本甚至比新版本还要稳定，这些公司都会停止对旧版本软件的支持，但他们也会因此而臭名昭著的。

像 Intuit 公司，在几年后就故意停止了对旧版本 Quicken 的支持。微软在其 Vista 系统还不稳定且没有普及时就已经停止对于 XP 系统的支持。更糟糕的是，Symantec 公司、Intuit 公司和微软公司往往会更改文件格式，这样一来新版本上的记录就无法被用于旧版本。客户们的不满最终引起了微软公司的重视，所以他们就提供了一些转换的方法。但 Intuit 公司和 Symantec 公司仍然未做出改变。

然而，在某种程度上，老化遗留程序是需要更换的。有时其中使用的硬件设备也是需要替换的。对于小型 PC 机和 Macintosh 系统，软件的更新换代还是比较方便的，其花费也不是难以承受的。然而，对于大多数的主流软件或拥有 10 000 个功能点左右的大型系统来说，更换就显得复杂而昂贵了。

如果软件是定制开发并且具有独特的功能，那么就需要开发出一个包含所有原始特性的新应用作为替代软件，并且还要增加一些有用的新功能。退伍军人管理局的病人记录系统就是老的遗留系统的一个例子，没有可用的商业替代软件。另一个有关更换老的遗留系统的问题就是，通常，遗留应用程序所使用的编程语言已经“死亡”而且没有可用的编译

器或解释器了，更不要说能够胜任这项工作的程序员，那就更少了。

对于即将退役的老系统（假设它们仍在使用中）来说，一般常用的最佳实践包括以下：

- 挖掘应用程序中的业务规则和算法，以备新应用之需。
- 对所有用户进行调查来决定应用程序业务操作的重要性。
- 通过互联网或咨询来对同种应用程序做深度调查。
- 在新版本更新过程中保持遗留程序的稳定和可用。
- 考虑面向服务架构（SOA）是否合适。
- 寻找可重用资料的认证来源。
- 考虑自动转换语言的可能性。
- 如果语言合适就使用静态分析工具。

毫无疑问，除非一个软件没有用户，否则更新和卸载该软件必然会给用户带来麻烦。

此外，一个值得注意的问题就是所有的储存形式的使用期限都是有限的。磁盘或固态设备都不可能在完全工作的模式下正常工作 25 年以上。

2.51 总结

我们可以得到以下 6 个最明显的结论：

第一，软件并不是一个“一刀切”的职业。需要有众多的实践和方法。

第二，不成熟的度量方法以及缺乏精确的量化数据都会导致评估操作难以进行。所幸的是由于具备可用的基准数据，这一状况目前正在得到改善。

第三，鉴于软件项目失败率增高、成本超支以及进度延期这些问题，正常的软件开发在经济上是不可持续的。从定制开发转向使用认证的可重用组件开发是实现软件开发经济化的必经之路。

第四，要想实现有效的软件重用，有效的质量控制是必经之路。将软件的缺陷预防、审查、静态分析、测试以及质量保证相结合也是必需的。

第五，随着软件安全威胁的数量以及严重性的增加，软件的架构、设计、编码方法以及防御方法需要做根本性的变化。

第六，大型软件应用程序的使用期限一般是 25 年或更长时间。其所使用的方法和一般做法不仅会在软件开发中用到，也会在接下来许多年的维护和改进中用到。

参考文献

第 2 章概述了很多不同的问题。出于实用角度的考虑，以下罗列了一些与本章所讨论问题相关的重要书籍和文献，而不是传统的参考列表。

项目管理、规划、评估、风险以及价值分析

Boehm, Barry Dr. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981. (中文版《软

- 件工程经济学》，李师贤等译，机械工业出版社2004年7月出版。）
- Booch Grady. *Object Solutions: Managing the Object-Oriented Project*. Reading, MA: Addison Wesley, 1995.
(中文版《面向对象项目的解决方案》，邢春丽、冯学民、张丽梅译，机械工业出版社2003年8月出版)
- Brooks, Fred. *The Mythical Man-Month*. Reading, MA: Addison Wesley, 1974, rev. 1995. (中文版《人月神话》，UMLChian 翻译组汪颖译，清华大学出版社出版)
- Charette, Bob. *Software Engineering Risk Analysis and Management*. New York: McGraw-Hill, 1989.
- Charette, Bob. *Application Strategies for Risk Management*. New York: McGraw-Hill, 1990.
- Chrissies, Mary Beth; Konrad, Mike; Shrum, Sandy; CMMI®: Guidelines for Product Integration and Process Improvement; Second Edition; Addison Wesley, Reading, MA; 2006; 704 pages.
- Cohn, Mike. *Agile Estimating and Planning*. Englewood Cliffs, NJ: Prentice Hall PTR, 2005. (中文版《敏捷估计与规划》，宋锐译，清华大学出版社2007年7月出版)
- DeMarco, Tom. *Controlling Software Projects*. New York: Yourdon Press, 1982.
- Ewusi-Mensah, Kwaku. *Software Development Failures* Cambridge, MA: MIT Press, 2003.
- Galarath, Dan. *Software Sizing, Estimating, and Risk Management: When Performance Is Measured Performance Improves*. Philadelphia: Auerbach Publishing, 2006.
- Glass, R.L. *Software Runaways: Lessons Learned from Massive Software Project Failures*. Englewood Cliffs, NJ: Prentice Hall, 1998.
- Harris, Michael, David Herron, and Stasia Iwanicki. *The Business Value of IT: Managing Risks, Optimizing Performance, and Measuring Results*. Boca Raton, FL: CRC Press (Auerbach), 2008.
- Humphrey, Watts. *Managing the Software Process*. Reading, MA: Addison Wesley, 1989. (中文版《软件过程管理》，高书敬、顾铁成、胡寅译，清华大学出版社2003年3月出版)
- Johnson, James, et al. *The Chaos Report*. West Yarmouth, MA: The Standish Group, 2000.
- Jones, Capers. *Assessment and Control of Software Risks*.: Prentice Hall, 1994.
- Jones, Capers. *Estimating Software Costs*. New York: McGraw-Hill, 2007. (中文版《软件项目估计》(第2版)，电子工业出版社，2008年3月出版)
- Jones, Capers. "Estimating and Measuring Object-Oriented Software." *American Programmer*, 1994.
- Jones, Capers. *Patterns of Software System Failure and Success*. Boston: International Thomson Computer Press, December 1995.
- Jones, Capers. *Program Quality and Programmer Productivity*. IBM Technical Report TR 02.764. San Jose, CA: January 1977.
- Jones, Capers. *Programming Productivity*. New York: McGraw-Hill, 1986.
- Jones, Capers. "Why Flawed Software Projects are not Cancelled in Time." *Cutter IT Journal*, Vol. 10, No. 12 (December 2003): 12-17.
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Boston: Addison Wesley Longman, 2000. (中文版《软件评估、基准测试与最佳实践》，机械工业出版社，2003年4月出版)
- Jones, Capers. "Software Project Management Practices: Failure Versus Success." *Crosstalk*, Vol. 19, No. 6 (June 2006): 4-8.
- Laird, Linda M. and Carol M. Brennan. *Software Measurement and Estimation: A Practical Approach*. Hoboken, NJ: John Wiley & Sons, 2006.

- McConnell, Steve. *Software Estimating: Demystifying the Black Art*. Redmond, WA: Microsoft Press, 2006.
- Park, Robert E., et al. *Software Cost and Schedule Estimating - A Process Improvement Initiative*. Technical Report CMU/SEI 94-SR-03. Pittsburgh, PA: Software Engineering Institute, May 1994.
- Park, Robert E., et al. *Checklists and Criteria for Evaluating the Costs and Schedule Estimating Capabilities of Software Organizations*. Technical Report CMU/SEI 95-SR-005. Pittsburgh, PA: Software Engineering Institute, Carnegie-Mellon Univ., January 1995.
- Roetzheim, William H. and Reyna A. Beasley. *Best Practices in Software Cost and Schedule Estimation*. Saddle River, NJ: Prentice Hall PTR, 1998.
- Strassmann, Paul. *Governance of Information Management: The Concept of an Information Constitution, Second Edition*. (eBook) Stamford, CT: Information Economics Press, 2004.
- Strassmann, Paul. *Information Productivity*. Stamford, CT: Information Economics Press, 1999.
- Strassmann, Paul. *Information Payoff*. Stamford, CT: Information Economics Press, 1985.
- Strassmann, Paul. *The Squandered Computer*. Stamford, CT: Information Economics Press, 1997.
- Stukes, Sherry, Jason Deshoretz, Henry Apgar, and Ilona Macias. *Air Force Cost Analysis Agency Software Estimating Model Analysis*. TR-9545/008-2 Contract F04701-95-D-0003, Task 008. Management Consulting & Research, Inc., Thousand Oaks, CA 91362. September 30, 1996.
- Symons, Charles R. *Software Sizing and Estimating—Mk II FPA (Function Point Analysis)*. Chichester, UK: John Wiley & Sons, 1991.
- Wellman, Frank. *Software Costing: An Objective Approach to Estimating and Controlling the Cost of Computer Software*. Englewood Cliffs, NJ: Prentice Hall, 1992.
- Whitehead, Richard. *Leading a Development Team*. Boston: Addison Wesley, 2001.
- Yourdon, Ed. *Death March - The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*. Upper Saddle River, NJ: Prentice Hall PTR, 1997. (中文版《死亡之旅》，周浩宇、杨华译，机械工业出版社 2012 年 1 月出版)
- Yourdon, Ed. *Outsource: Competing in the Global Productivity Race*. Upper Saddle River, NJ: Prentice Hall PTR, 2005.

度量指标

- Abran, Alain and Reiner R. Dumke. *Innovations in Software Measurement*. Aachen, Germany: Shaker-Verlag, 2005.
- Abran, Alain, Manfred Bundschuh, Reiner Dumke, Christof Ebert, and Horst Zuse. ["article title"?] *Software Measurement News*, Vol. 13, No. 2 (Oct. 2008). (periodical).
- Bundschuh, Manfred and Carol Dekkers. *The IT Measurement Compendium*. Berlin: Springer-Verlag, 2008.
- Chidamber, S. R. and C. F. Kemerer. "A Metrics Suite for Object-Oriented Design," *IEEE Trans. On Software Engineering*, Vol. SE20, No. 6 (June 1994): 476-493.
- Dumke, Reiner, Rene Braungarten, Günter Büren, Alain Abran, Juan J. Cuadrado-Gallego, (editors). *Software Process and Product Measurement*. Berlin: Springer-Verlag, 2008.
- Ebert, Christof and Reiner Dumke. *Software Measurement: Establish, Extract, Evaluate, Execute*. Berlin: Springer-Verlag, 2007.

- Garmus, David & David Herron. *Measuring the Software Process: A Practical Guide to Functional Measurement*. Englewood Cliffs, NJ: Prentice Hall, 1995.
- Garmus, David and David Herron. *Function Point Analysis – Measurement Practices for Successful Software Projects*. Boston: Addison Wesley Longman, 2001.
- International Function Point Users Group. *IFPUG Counting Practices Manual*, Release 4. Westerville, OH: April 1995.
- International Function Point Users Group (IFPUG). *IT Measurement – Practical Advice from the Experts*. Boston: Addison Wesley Longman, 2002.
- Jones, Capers. *Applied Software Measurement, Third Edition*. New York: McGraw-Hill, 2008.
- Jones, Capers. "Sizing Up Software." *Scientific American Magazine*, Vol. 279, No. 6 (December 1998): 104–111.
- Jones Capers. *A Short History of the Lines of Code Metric*, Version 4.0. (monograph) Narragansett, RI: Capers Jones & Associates LLC, May 2008.
- Kemerer, C. F. "Reliability of Function Point Measurement – A Field Experiment." *Communications of the ACM*, Vol. 36, 1993: 85–97.
- Parthasarathy, M. A. *Practical Software Estimation – Function Point Metrics for Insourced and Outsourced Projects*. Upper Saddle River, NJ: Infosys Press, Addison Wesley, 2007.
- Putnam, Lawrence H. *Measures for Excellence – Reliable Software On Time, Within Budget*. Englewood Cliffs, NJ: Yourdon Press – Prentice Hall, 1992.
- Putnam, Lawrence H. and Ware Myers. *Industrial Strength Software – Effective Management Using Measurement*. Los Alamitos, CA: IEEE Press, 1997.
- Stein, Timothy R. *The Computer System Risk Management Book and Validation Life Cycle*. Chico, CA: Paton Press, 2006.
- Stutzke, Richard D. *Estimating Software-Intensive Systems*. Upper Saddle River, NJ: Addison Wesley, 2005.

架构、需求以及设计

- Ambler, S. *Process Patterns – Building Large-Scale Systems Using Object Technology*. Cambridge University Press, SIGS Books, 1998.
- Artow, J. and I. Neustadt. *UML and the Unified Process*. Boston: Addison Wesley, 2000.
- Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Boston: Addison Wesley, 1997.
- Berger, Arnold S. *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*. CMP Books, 2001.
- Booch, Grady, Ivar Jacobsen, and James Rumbaugh. *The Unified Modeling Language User Guide*, Second Edition. Boston: Addison Wesley, 2005.
- Cohn, Mike. *User Stories Applied: For Agile Software Development*. Boston: Addison Wesley, 2004.
- Fernandini, Patricia L. *A Requirements Pattern Succeeding in the Internet Economy*. Boston: Addison Wesley, 2002.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable*

- Object Oriented Design*. Boston: Addison Wesley, 1995.
- Inmon William H., John Zachman, and Jonathan G. Geiger. *Data Stores, Data Warehousing, and the Zachman Framework*. New York: McGraw-Hill, 1997.
- Marks, Eric and Michael Bell. *Service-Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology*. New York: John Wiley & Sons, 2006.
- Martin, James & Carma McClure. *Diagramming Techniques for Analysts and Programmers*. Englewood Cliffs, NJ: Prentice Hall, 1985.
- Orr, Ken. *Structured Requirements Definition*. Topeka, KS: Ken Orr and Associates, Inc, 1981.
- Robertson, Suzanne and James Robertson. *Mastering the Requirements Process, Second Edition*. Boston: Addison Wesley, 2006.
- Warnier, Jean-Dominique. *Logical Construction of Systems*. London: Van Nostrand Reinhold.
- Wieggers, Karl E. *Software Requirements*, Second Edition. Bellevue, WA: Microsoft Press, 2003.

软件质量控制

- Beck, Kent. *Test-Driven Development*. Boston: Addison Wesley, 2002.
- Chelf, Ben and Raoul Jetley. "Diagnosing Medical Device Software Defects Using Static Analysis." Coverity Technical Report. San Francisco: 2008.
- Chess, Brian and Jacob West. *Secure Programming with Static Analysis*. Boston: Addison Wesley, 2007.
- Cohen, Lou. *Quality Function Deployment – How to Make QFD Work for You*. Upper Saddle River, NJ: Prentice Hall, 1995.
- Crosby, Philip B. *Quality is Free*. New York: New American Library, Mentor Books, 1979.
- Everett, Gerald D. and Raymond McLeod. *Software Testing*. Hoboken, NJ: John Wiley & Sons, 2007.
- Gack, Gary. *Applying Six Sigma to Software Implementation Projects*. <http://software.isixsigma.com/library/content/c040915b.asp>.
- Gilb, Tom and Dorothy Graham. *Software Inspections*. Reading, MA: Addison Wesley, 1993.
- Hallowell, David L. *Six Sigma Software Metrics, Part I*. <http://software.isixsigma.com/library/content/03910a.asp>.
- International Organization for Standards. "ISO 9000 / ISO 14000." <http://www.iso.org/iso/en/iso9000-14000/index.html>.
- Jones, Capers. *Software Quality – Analysis and Guidelines for Success*. Boston: International Thomson Computer Press, 1997.
- Kan, Stephen H. *Metrics and Models in Software Quality Engineering, Second Edition*. Boston: Addison Wesley Longman, 2003.
- Land, Susan K., Douglas B. Smith, John Z. Walz. *Practical Support for Lean Six Sigma Software Process Definition: Using IEEE Software Engineering Standards*. Wiley-Blackwell, 2008.
- Mosley, Daniel J. *The Handbook of MIS Application Software Testing*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall, 1993.
- Myers, Glenford. *The Art of Software Testing*. New York: John Wiley & Sons, 1979. (最新中文版《软件测试的艺术》(原书第3版), 张晓明、黄琳译, 机械工业出版社2012年4月出版。)

Nandyal Raghav. *Making Sense of Software Quality Assurance*. New Delhi: Tata McGraw-Hill Publishing, 2007.

Radice, Ronald A. *High Quality Low Cost Software Inspections*. Andover, MA: Paradoxicon Publishing, 2002.

Wieggers, Karl E. *Peer Reviews in Software – A Practical Guide*. Boston: Addison Wesley Longman, 2002. (中文版《软件同级评审》，沈备军、宿为民译，机械工业出版社2003年6月出版。)

软件安全、黑客以及恶意软件的预防

Acohido, Byron and John Swartz. *Zero Day Threat: The Shocking Truth of How Banks and Credit Bureaus Help Cyber Crooks Steal Your Money and Identity.*: Union Square Press, 2008.

Allen, Julia, Sean Barnum, Robert Ellison, Gary McGraw, and Nancy Mead. *Software Security: A Guide for Project Managers*. (An SEI book sponsored by the Department of Homeland Security) Boston: Addison Wesley Professional, 2008.

Anley, Chris, John Heasman, Felix Lindner, and Gerardo Richarte. *The Shellcoders Handbook: Discovering and Exploiting Security Holes*. New York: Wiley, 2007.

Chess, Brian. *Secure Programming with Static Analysis*. Boston: Addison Wesley Professional, 2007.

Dowd, Mark, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Boston: Addison Wesley Professional, 2006.

Ericson, John. *Hacking: The Art of Exploitation*, Second Edition.: No Starch Press, 2008.

Gallager, Tom, Lawrence Landauer, and Brian Jeffries. *Hunting Security Bugs*. Redmond, WA: Microsoft Press, 2006.

Hamer-Hodges, Ken. *Authorization Oriented Architecture – Open Application Networking and Security in the 21st Century*. Philadelphia: Auerbach Publications, to be published in December 2009.

Hogland, Greg and Gary McGraw. *Exploiting Software: How to Break Code*. Boston: Addison Wesley Professional, 2004.

Hogland, Greg and Jamie Butler. *Rootkits: Exploiting the Windows Kernel*. Boston: Addison Wesley Professional, 2005.

Howard, Michael and Steve Lippner. *The Security Development Lifecycle*. Redmond, WA: Microsoft Press, 2006.

Howard, Michael and David LeBlanc. *Writing Secure Code*. Redmond, WA: Microsoft Press, 2003.

Jones, Andy and Debi Ashenden. *Risk Management for Computer Security: Protecting Your Network and Information Assets.*: Butterworth-Heinemann, 2005.

Landoll, Douglas J. *The Security Risk Assessment Handbook: A Complete Guide for Performing Security Risk Assessments*. Boca Raton, FL: CRC Press (Auerbach), 2005.

McGraw, Gary. *Software Security – Building Security In*. Boston: Addison Wesley Professional, 2006.

Rice, David. *Geekonomics: The Real Cost of Insecure Software*. Boston: Addison Wesley Professional, 2007.

Scambray, Joel. *Hacking Exposed Windows: Microsoft Windows Security Secrets and Solutions, Third Edition*. New York: McGraw-Hill, 2007.

——— *Hacking Exposed Web Applications*, Second Edition. New York: McGraw-Hill, 2006.

Sherwood, John, Andrew Clark, and David Lynas. *Enterprise Security Architecture: A Business-Driven Approach.*: CMP, 2005.

- Shostack, Adam and Andrews Stewart. *The New School of Information Security*. Boston: Addison Wesley Professional, 2008.
- Skudis, Edward and Tom Liston. *Counter Hack Reloaded: A Step-by-Step Guide to Computer Attacks and Effective Defenses*. Englewood Cliffs, NJ: Prentice Hall PTR, 2006.
- Skudis, Edward and Lenny Zeltzer. *Malware: Fighting Malicious Code*. Englewood Cliffs, NJ: Prentice Hall PTR, 2003.
- Stuttard, Dafydd and Marcus Pinto. *The Web Application Hackers Handbook: Discovering and Exploiting Security Flaws*. New York: Wiley, 2007.
- Szor, Peter. *The Art of Computer Virus Research and Defense*. Boston: Addison Wesley Professional, 2005.
- Thompson, Herbert and Scott Chase. *The Software Vulnerability Guide*. Boston: Charles River Media, 2005.
- Viega, John and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston: Addison Wesley Professional, 2001.
- Whittaker, James A. and Herbert H. Thompson. *How to Break Software Security*. Boston: Addison Wesley Professional, 2003.
- Wysopal, Chris, Lucas Nelson, Dino Dai Zovi, and Elfriede Dustin. *The Art of Software Security Testing: Identifying Software Security Flaws*. Boston: Addison Wesley Professional, 2006.

软件工程和编码

- Barr, Michael and Anthony Massa. *Programming Embedded Systems: With C and GNU Development Tools*. O'Reilly Media, 2006.
- Beck, K. *Extreme Programming Explained: Embrace Change*. Boston: Addison Wesley, 1999.
- Bott, Frank, A. Coleman, J. Eaton, and D. Roland. *Professional Issues in Software Engineering*. Taylor & Francis, 2000.
- Glass, Robert L. *Facts and Fallacies of Software Engineering (Agile Software Development)*. Boston: Addison Wesley, 2002.
- Hans, Professor van Vliet. *Software Engineering Principles and Practices*, Third Edition. London, New York: John Wiley & Sons, 2008.
- Hunt, Andrew and David Thomas. *The Pragmatic Programmer*. Boston: Addison Wesley, 1999.
- Jeffries, R., et al. *Extreme Programming Installed*. Boston: Addison Wesley, 2001.
- Marciniak, John J. (Editor). *Encyclopedia of Software Engineering* (two volumes). New York: John Wiley & Sons, 1994.
- McConnell, Steve. *Code Complete*. Redmond, WA: Microsoft Press, 1993.
- Morrison, J. Paul. *Flow-Based Programming: A New Approach to Application Development*. New York: Van Nostrand Reinhold, 1994.
- Pressman, Roger. *Software Engineering – A Practitioner's Approach*, Sixth Edition. New York: McGraw-Hill, 2005.
- Sommerville, Ian. *Software Engineering. Seventh Edition*. Boston: Addison Wesley, 2004.
- Stephens M. and D. Rosenberg. *Extreme Programming Refactored: The Case Against XP*. Berkeley, CA: Apress L.P., 2003.

软件开发方法

- Boehm, Barry. "A Spiral Model of Software Development and Enhancement." Proceedings of the Int. Workshop on Software Process and Software Environments. *ACM Software Engineering Notes* (Aug. 1986): 22-42.
- Cockburn, Alistair. *Agile Software Development*. Boston: Addison Wesley, 2001.
- Cohen, D., M. Lindvall, and P. Costa. "An Introduction to agile methods." *Advances in Computers*, New York: Elsevier Science, 2004.
- Highsmith, Jim. *Agile Software Development Ecosystems*. Boston: Addison Wesley, 2002.
- Humphrey, Watts. *TSP - Leading a Development Team*. Boston: Addison Wesley, 2006.
- Humphrey, Watts. *PSP: A Self-Improvement Process for Software Engineers*. Upper Saddle River, NJ: Addison Wesley, 2005.
- Krutchén, Phillippe. *The Rational Unified Process - An Introduction*. Boston: Addison Wesley, 2003.
- Larman, Craig and Victor Basili. "Iterative and Incremental Development - A Brief History." *IEEE Computer Society* (June 2003): 47-55.
- Love, Tom. *Object Lessons*. New York: SIGS Books, 1993.
- Martin, Robert. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice Hall, 2002.
- Mills, H., M. Dyer, and R. Linger. "Cleanroom Software Engineering." *IEEE Software*, 4, 5 (Sept. 1987): 19-25.
- Paulk Mark, et al. *The Capability Maturity Model Guidelines for Improving the Software Process*. Reading, MA: Addison Wesley, 1995.
- 快速应用开发. http://en.wikipedia.org/wiki/Rapid_application_development.
- Stapleton, J. *DSDM - Dynamic System Development Method in Practice*. Boston: Addison Wesley, 1997.

软件部署、客户支持以及维护

- Arnold, Robert S. *Software Reengineering*. Los Alamitos, CA: IEEE Computer Society Press, 1993.
- Arthur, Lowell Jay. *Software Evolution - The Software Maintenance Challenge*. New York: John Wiley & Sons, 1988.
- Gallagher, R. S. *Effective Customer Support*. Boston: International Thomson Computer Press, 1997.
- Parikh, Girish. *Handbook of Software Maintenance*. New York: John Wiley & Sons, 1986.
- Pigoski, Thomas M. *Practical Software Maintenance - Best Practices for Managing Your Software Investment*. Los Alamitos, CA: IEEE Computer Society Press, 1997.
- Sharon, David. *Managing Systems in Transition - A Pragmatic View of Reengineering Methods*. Boston: International Thomson Computer Press, 1996.
- Takang, Armstrong and Penny Grubb. *Software Maintenance Concepts and Practice*. Boston: International Thomson Computer Press, 1997.
- Ulrich, William M. *Legacy Systems: Transformation Strategies*. Upper Saddle River, NJ: Prentice Hall, 2002.

软件工程中的社交

- Brooks, Fred. *The Mythical Manmonth*, Second Edition. Boston: Addison Wesley, 1995.

- DeMarco, Tom. *Peopleware: Productive Projects and Teams*. New York: Dorset House, 1999.
- Glass, Robert L. *Software Creativity*, Second Edition. Atlanta, GA: developer.*books, 2006.
- Humphrey, Watts. *Winning with Software: An Executive Strategy*. Boston: Addison Wesley, 2002.
- Johnson, James, et al. *The Chaos Report*. West Yarmouth, MA: The Standish Group, 2007.
- Jones, Capers. "How Software Personnel Learn New Skills," Sixth Edition (monograph). Narragansett, RI: Capers Jones & Associates LLC, July 2008.
- Jones, Capers. "Conflict and Litigation Between Software Clients and Developers" (monograph). Narragansett, RI: Software Productivity Research, Inc., 2008.
- Jones, Capers. "Preventing Software Failure: Problems Noted in Breach of Contract Litigation." Narragansett, RI: Capers Jones & Associates LLC, 2008.
- Krasner, Herb. "Accumulating the Body of Evidence for the Payoff of Software Process Improvement - 1997" Austin, TX: Krasner Consulting.
- Kuhn, Thomas. *The Structure of Scientific Revolutions*. University of Chicago Press, 1996.
- Starr, Paul. *The Social Transformation of American Medicine*.: Basic Books Perseus Group, 1982.
- Weinberg, Gerald M. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971. (最新中文版《程序开发心理学》(银周年纪念版), 电子工业出版社 2010 年 3 月出版)
- Weinberg, Gerald M. *Becoming a Technical Leader*. New York: Dorset House, 1986.
- Yourdon, Ed. *Death March - The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*. Upper Saddle River, NJ: Prentice Hall PTR, 1997.
- Zoellick, Bill. *CyberRegs - A Business Guide to Web Property, Privacy, and Patents*. Boston: Addison Wesley, 2002.

网站

目前, 有数百个软件行业和专业协会, 它们中的大多数都有一个狭窄研究范围。它们中的大多数都或多或少地是特立独行的, 并且没有接触过相似的协会。不过也有例外的, 如各种软件过程改进网络 (SPIN) 组织以及各种软件度量协会。

部分软件组织和网站是为了方便交流和共享数据的, 一般情况下会有组织和国家的限制。软件是一个全球性的行业。软件从需求阶段开始, 一直到最后交付用户使用, 这之间的每天都会出现问题。因此, 打破行业和技术限制, 相互合作将有利于整个行业的发展, 并促使软件行业成为一门真正的职业, 而不是一门边缘的艺术。

对软件产业来说, 最有用的就是像 AMA (美国医学协会) 一样, 各个主要的专业协会相互认可对方会员的资格。作为一种职业, 软件行业有必要成立一个伞式组织, 该组织涉及软件的方方面面, 就像 AMA 的医疗实践一样。

American Electronics Association (AEA) www.aeanet.org (may merge with ITAA)

American Society for Quality www.ASQ.org

Anti-Phishing Working Group www.antiphishing.org

Association for Software Testing www.associationforsoftwaretesting.org

Association of Computing Machinery www.ACM.org

Association of Competitive Technologies (ACT) www.actonline.org

Association of Information Technology Professionals www.aitp.org
Brazilian Function Point Users Group www.BFPUG.org
Business Application Software Developers Association www.basda.org
Business Software Alliance (BSA) www.bsa.org
Center for Internet Security www.cisecurity.org
Center for Hybrid and Embedded Software Systems (CHESS) <http://chess.eecs.berkeley.edu>
China Software Industry Association www.CSIA.org
Chinese Software Professional Association www.CSPA.com
Computing Technology Industry Association (CTIA) www.comptia.org
Embedded Software Association (ESA) www.esofta.com
European Design and Automation Association (EDAA) www.edaa.com
Finnish Software Measurement Association www.fisma.fi
IEEE Computer Society www.computer.org
Independent Computer Consultants Association (ICCA) www.icca.org
Information Technology Association of America (ITAA) www.itaa.org (may merge with AEA)
Information Technology Metrics and Productivity Institute (ITMPI) www.ITMPI.org
InfraGuard www.InfraGuard.net
Institute for International Research (IIR) eee.irusa.com
Institute of Electrical and Electronics Engineers (IEEE) www.IEEE.org
International Association of Software Architects www.IASAHOME.org
International Function Point Users Group (IFPUG) www.IFPUG.org
International Institute of Business Analysis www.IIBAorg
International Software Benchmarking Standards Group (ISBSG) www.ISBSG.org
Japan Function Point Users Group www.jfpug.org
Linux Professional Institute www.lpi.org
National Association of Software and Service Companies (India) www.NASCOM.in
Netherlands Software Metrics Association www.NESMA.org
Process Fusion www.process-fusion.com
Programmers' Guild www.programmersguild.org
Project Management Institute www.PMI.org
Russian Software Development Organization (RUSOFT) www.russoft.org
Society of Information Management (SIM) www.simnet.org
Software and Information Industry Association www.siia.net
Software Engineering Body of Knowledge www.swebok.org
Software Engineering Institute (SEI) www.SEI.org
Software Productivity Research (SPR) www.SPR.com
Software Publishers Association (SPA) www.spa.org
United Kingdom Software Metrics Association www.UKSMA.org
U.S. Internet Industry Association (USIIA) www.usiia.org
Women in Technology International www.witi.com

2049 年的软件开发和维护预览

3.1 引言

从 20 世纪 60 年代到 2009 年，软件开发基本上一直是一门工艺，复杂的应用被设计成独一无二的工件，之后再通过手工编码来逐行构建。虽然这种个性化的开发方法使用自定义的手工编码，然而这样的方法既不高效也不经济，甚至不可能使软件达到一致的质量或者安全水平。

在油画布上创作和画一幅肖像与开发一个软件应用在本质上是很相似的。每一件艺术品都是独一无二的，并且通过使用个人的画笔来创作；要使画作的整体效果具有美感，并给人留下深刻的印象，我们就需要完美有序地放置和组织这些画笔。然而，不论是肖像还是软件应用都不是工程学科。

希望到了 2049 年，一门真正的工程学科会出现，即软件从一种艺术表达形式演变成一门可靠的工程学科。本章介绍了 2049 年前后的一种假设性分析，即软件应用的设计和构造方式。

如果软件工程应该成为一门真正的工程学科，那么软件工程要包含的就不仅是代码开发了，还需要考虑架构、需求、设计、代码开发、维护、客服支持、培训、文档、指标和度量、项目管理、安全、质量、变更控制、基准以及其他主题。

不管是 2009 年还是 2049 年，对于新应用来说，需求理所当然是开发的第一步。在 2009 年，通过用户访谈来获得新应用的需求；但是到了 2049 年，或许可以使用一种全新的方法了。

让我们这么假设，大约 2049 年，待开发的应用是一种新型的软件规划和成本评估工具。这种工具将会和现有的一些工具一样提供软件成本评估、进度评估、质量评估以及员工评估等。然而，这种工具也会推出一些新的功能，如：

1. 在获得完整的需求之前及时估算应用的规模。
2. 在开发期间，对需求变更的评估。
3. 蔓延需求中的缺陷数量的评估。
4. 综合风险分析。
5. 综合价值分析。

6. 综合安全分析。
7. 预测任何 CMMI 等级对生产率和质量的影响。
8. 预测不同数量的可重用材料的影响。
9. 预测智能工具对软件开发的影响。
10. 预测智能工具对软件维护的影响。
11. 预测智能工具对软件文档的影响。
12. 预测智能工具对软件客户支持的影响。
13. 预测智能工具对软件故障的影响。
14. 功能点、LOC (代码行)、故事点^①之间的自动转换。
15. 对部分的用户学习曲线进行评估。
16. 当用户学习应用的时候, 对用户犯的错误进行评估。
17. 在软件部署 10 年之后, 对客户的支持以及软件的维护进行评估。
18. 在软件部署 10 年之后, 对应用的增长进行评估。
19. 在开发和维护期间, 综合采集历史数据。
20. 自动创建生产率和质量基准。
21. 软件质量控制方面的专家建议。
22. 软件安全控制方面的专家建议。
23. 软件治理方面的专家建议。
24. 知识产权方面的专家建议。
25. 相关标准和法规的专家建议。

评估工具的第 19 和第 20 个新功能将会涉及建立一种由 ISBSG (国际软件基准组织) 认证的整体授权, 客户在评估新应用的时候, 就能够通过该工具来聚合和分析相似应用的基准。每一个客户都需要为该服务支付一定的费用, 但是这项服务不应该集成在这个工具里面。这样, 评估不仅可以通过工具来产生, 而且也可以通过提供相似应用的历史数据, 以聚合相似应用的基准, 并且这些基准也可以用来支持评估。

这个新式评估工具是用来收集历史数据的, 并且半自动地创建基准。这些基准将会使用 ISBSG 问题集, 并且对于具体的专题, 还包括一些额外的问题, 例如安全、缺陷去除效率以及客户支持等, 但是在 ISBSG 中却不包括客户支持。

由于该工具将被用来预测, 并且存储机密的分类信息, 因此对于该工具来说, 安全是一个苛刻的要求, 除此之外, 也会向该工具添加一些安全功能, 例如加密所有存储信息等。

我们还可以假设, 构建新评估工具的公司在同行业中至少已经生产了一种先进的工具; 换句话说, 就是现有的产品在公司内部可以用来分析。

3.2 需求分析

在 2049 年, 收集需求的第一步将会是, 让智能工具或虚拟化身从 Web 上提取与软件评

^① 故事点是敏捷开发中所使用的一种随机度量方式, 用来度量实现一个故事需要付出的工作量。

估和规划工具相关的所有信息。所有的技术文献和市场信息都将会通过相似的工具来收集和分析,例如,Application Insight、Artemis Views、Checkpoint、COCOMO 和它的复制品、KnowledgePlan、Microsoft Project、Price-S、SEER、SLIM、SPQR/20、SoftCost 以及其他一些工具。

在所有相似的工具中,智能工具也会产生一个综合清单,清单中会罗列所有目前可用的功能,包括调整规模的方法、货币换算、通胀率的调整、质量预测、总拥有成本等。

希望到 2049 年,软件重用已经达到了一定的成熟度,可以提供全面的可重用构件目录;质量和安全的认证将是司空见惯的;架构和设计将达到一个新的高度;应用程序、连接点以及其他相关问题的标准结构描述将很容易获取。

智能工具也将会从公共记录中收集信息,包括工具已出售的副本个数、工具的收益、工具对应的用户协会、对工具供应商的诉讼以及其他相关的业务主题。

如果这个工具被用来评估财务软件应用,智能工具还会从网上扫描所有可能适用的政府法规,例如《萨班斯-奥克斯利法案》(Sarbanes-Oxley)以及其他相关的规定。由于金融危机和经济衰退,大量新法规即将浮出水面,并且只有智能工具和专家系统能够跟上步伐,及时获取相关信息。

对于其他形式的软件,智能工具也许会从网上扫描法规、标准以及其他的主题,这些主题影响了治理,并且还影响了政府的授权,例如,处理医疗设备和处理医疗记录的软件应用或者需要法律上保护隐私的软件应用等。

一旦领域中现存的工具和功能集被分析出来,那么下一步就是考虑增加新功能的价值,让它的价值超过现有项目中已经使用的规划和评估工具。2049 年的需求将类似于 2009 年,在需求中,将收集并分析众多投资者的意见和想法。

由于将开发的应用程序是一个专家系统,因此多数关于新功能的信息必须来自软件规划和评估的专家。尽管通过调查或者焦点小组得到的客户意见是有用的,并且通过市场营销组织得到的意见也是有用的,但是只有专家才有资格指定比较独特的功能。

话虽然这么说,但是当新功能的需求正在计划中时,另一个平行的工作也得展开了,这就是为部分或者全部新功能进行专利申请。同样,在这里,将会派遣一个智能工具去收集和分析所有覆盖了新功能的专利,这些专利也许与新型评估工具的设计相似。

假设大多数的新功能真的是独一无二的,并且在任何现有的评估工具中都不存在,那么就需要为该需求准备大约 6~20 项应用专利,然后再进行组装。在构建应用程序的时候,一个重要的步骤是包含新的知识内容:违反专利可能会造成重大损失,并且彻底地停止开发。我们要特别注意专利公司,例如知识产权风险投资公司,他们的主要业务就是专利授权。

除了专利之外,也可能有商业机密、发明披露、版权以及其他形式的机密、专有信息、算法的保护等。

对于在这个例子中讨论的工具,及早估算规模的功能、在开发中预测需求变更的功能、预测客户学习曲线成本的功能都将需要专利保护。其他的主题也许也需要专利保护,但是刚才提到的三个功能都是新颖独特的,在竞争性的工具中不曾出现过的。例如,目前没有

任何评估工具的算法涉及智能工具的影响。

需求分析阶段也需要研究新型评估工具的运行平台；这就是说，什么样的操作系统和什么样的硬件平台对应什么样的工具。毫无疑问，对个人电脑来说，拥有这种性质的工具将是一个很好的候选，但是也许可以只开发该功能的一个子集，用在手持设备上。在任何情况下，这类工具可能会运行在多个平台上，因此需要为 Windows、Apple、Linux、Unix 以及其他一些平台做规划。

这个工具不仅在多个平台上操作，而且很明显，该工具在很多国家都会有价值。同样，在这里，会派遣一个智能工具去寻找类似的工具，这个工具适合中国、日本、俄罗斯、韩国、巴西、墨西哥以及其他一些国家。这些信息将会成为营销规划的一部分，也将会用来确定必须建立多少个版本，并且会将信息翻译成其他的自然语言。

在当前的市场规模下，通过智能工具来收集信息，需求分析的其他方面将是预测新工具和新功能在客户、收入以及竞争优势等方面的市场潜力。对于任何公司，新功能将产生巨大的收益，可能是新产品开发和维护成本的 10 倍以上。

需求阶段的产出包括新工具的需求、所有相关应用领域的专利数据汇总以及目前市场上的评估工具和项目规划工具的汇总，并且这些工具在所有国家都能产生显著收益。当然，政府相关法规的汇总也是必需的。有趣的是，这些产出的 85% 都可以通过智能工具和专家系统来获得，并且只需要少量的人力劳动即可，即设置搜索条件。

从表面上看，设计成 SOA（面向服务架构）的应用也可以设想成标准可重用组件的集合。OO（面向对象）模式纳入可重用对象已经超过 30 年了。然而，无论是 SOA 还是 OO 模式都没有对遗留应用的算法和业务规则进行挖掘。它们都没有使用智能工具来搜索网页。SOA 和 OO 都没有把开发的新功能作为可重用对象，尽管 OO 模式很接近。此外，SOA 方法和 OO 方法都没有和真正安全的应用一样拥有严格的质量控制和安全实践。例如，在这两个领域里，可重用代码的认证也是参差不齐的。

3.3 设计

由于已经存在许多相似的应用，并且公司也已经构建了相似的应用，因此设计并不需要从头开始。相反，设计应该从详细的架构分析以及所有相似应用的设计开始。

2009 年的设计与 2049 年的设计有一个重要差异，就是许多可重用功能的使用，这些功能可能来自内部资源、商业资源，也可能来自认证的可重用功能库。

例如，由于应用程序是一个成本评估工具，因此，毫无疑问，无论是商业供应商开发的工具还是内部开发工具，货币转换、通胀率调整、内部会计收益率以及许多其他功能都可以提供可重用的形式。

一些打印输出也许会使用报告生成工具，如水晶报表（Crystal Reports）或类似的工具。一些应用程序数据也许可存储在正规商业数据库中，例如 Access、Bento 等。

由于公司正在构建的应用已经有相似的应用，因此毫无疑问，许多质量评估、进度评估以及基本成本评估等功能将会适用于正在构建的应用。需要注意的是，要想在经济上成

功,可重用的认证必须达到零缺陷的水平才行。

在理想的情况下,至少有 85% 的功能和设计元素会以可重用的形式出现,并且只有 15% 是真正需要定制设计的。对于新功能,确保高水平的质量和安全是很重要的,因此在添加新功能之后,需要执行设计审查。

然而,单个应用程序的定制开发是从来没有成本效益的。因此,2009 年的设计和 2049 年的设计有一个重要差异,那就是几乎所有的新功能都会被设计成可重用的组件,而不是被设计成单个应用程序的独特构件。

除了正规复用作为所有重要功能的设计目标,安全、质量、跨平台(Windows、Apple、Linux、UNIX 等)的可移植性都是基本的设计目标。就像一个普通的实践一样,应该淘汰单个应用的定制设计,并且替换成支持多应用和跨平台的可重用设计。

例如,允许在不知道所有需求的情况下,新功能在早期开发时是可以调整规模的,显然这个新功能可能会授权给其他公司或者供其他应用程序使用。因此,新功能需要适用于多种用途和跨平台。当然,新功能也需要专利保护。

2049 年的设计环境将可能会完全不同于 2009 年的设计环境。例如,由于大多数的应用都是基于先前的应用,因此会从传统的应用中提取先前所有功能的描述。假设过去的描述一直没有完全更新,甚至丢失,那么通过在源代码中进行数据挖掘,我们可以自动地从遗留代码中提取设计和算法。

因此,在将来,软件设计师可以将精力放在创新上,而不是放在调整遗留应用中。从遗留应用带入设计的功能将会通过专家系统产生,也会通过智能工具在网上搜索相似的应用而来。

为了从相似的遗留应用中挖掘信息,必须要有专家系统的设计工具。这个工具将会包括如下功能:静态分析、复杂度分析、安全分析、架构和设计结构分析,以及从遗留代码中提取算法和业务规则的能力。

工具的输出将包括结构设计图、控制流信息以及无用代码,也包括嵌入在遗留代码中的业务规则和算法的文本和数学描述。

通过审查 Web 和已发表文献中所提供的信息,甚至可以通过智能工具来自动构建用例和“用户故事”。只需要投入少量的人力,所有应用的数据字典也都能够通过专家系统来构建。

由于软件是动态的,因此可以预测,在 2049 年,动画和模拟也将成为设计的一部分。也可能创建应用的 3D 动态模型来处理例如性能、安全漏洞以及质量等问题,而这些问题在纸张上使用静态的表现形式并不容易理解。

由于通过智能工具和设计引擎可以自动地完成大多数的工作,因此完整的设计将会展示新旧两方面的功能,甚至也包括新应用和竞争性应用之间的比较。对于少数新功能的算法,需要由专家进行手工设计和构造,例如早期估算项目规模、需求增长、客户学习曲线等。

软件工程要成为一门真正的工程学科,我们有必要采用有效的方法来分析 and 确定软件应用的优化设计。将每个应用设计成一个独特的定制产品并不是一门真正的工程。一个专家系统能够分析现有应用的结构、功能、性能以及易用性,这是软件从一门艺术转变成一门工程学科的基本组成部分。

实际上,到了2049年,会出现数以百计的优化设计目录,包含了软件架构和设计的标准功能,而认证的可重用组件目录也包括在内。要做到这一点,就需要对应用程序的类型和功能进行分类。此外,标准的架构结构必须符合某种方法,并且可能遵循 Zachman 架构方法。

3.4 软件开发

假设软件应用85%的功能都以标准可重用组件的形式出现,那么2049年的软件开发将会与当前独特应用的逐行编码大相径庭。

到了2049年,软件开发的第一步将是积累所有已存在的可重用组件,并把它们一块儿放到工作原型中,同时也为今后将要添加的新功能提供占位符。该原型可以用来评估基本的问题,例如易用性、性能、安全性以及质量等。

由于新的功能已经经过创建和测试,因此它们可以被附加到原始的工作原型中。这条途径有些类似于敏捷开发,然而,敏捷开发在多数情况下并不是从挖掘遗留应用的数据开始的。

敏捷开发的一些后勤工作对我们也是很有用的,例如日常的进度会议或者 Scrum 会议。

但是,由于开发的目的是构建可重用对象而不是独一无二的单用对象,因此也会使用其他重视并且度量质量的技术。例如,团队软件过程和个体软件过程方法已经展示了非常高的质量控制水平。

由于新应用有非常严格的安全和质量要求,因此这些可重用组件必须经过认证,达到零缺陷的水平才行。如果不能提供这样的认证,那么候选的可重用组件必须通过一系列非常全面的检查,这些检查包括自动静态分析、动态分析、测试,或许还应包括审查。除此之外,还会收集并分析所有可重用组件的历史记录,以评估任何先前可能已经报道过的质量和安全漏洞。

由于应用的新功能并不打算设计成单用,而是打算设计成可重用组件,因此,很明显,开发这些组件就需要格外仔细。对于新功能所使用的开发方法,团队软件过程和个体软件过程对创建可重用构件似乎有严格要求。像敏捷开发或者其他途径的一些后勤方法都可以使用,但是严格和高质量水平是成功重用的主要目标。

由于需要高质量的组件,因此自动的静态和动态分析、仔细测试、现场检查等方法都是必需的。特别是,特殊类型的审查也是必需的,如专注于安全漏洞和缺陷的审查。

由于安全问题,例如支持安全的E语言可能会用来开发。然而,一些旧的可重用组件毫无疑问是用其他的语言编写的,例如C、Java、C++等,因此可能需要进行语言转换。然而,希望到了2049年,针对任何一门语言,所有可重用组件都有一个对应的安全版本。

例子中讨论了一种类型的软件成本评估应用,在2009年时它一般只有约2500个功能点。构建并实现这些应用通常需要两年半的时间,生产率约为每人每月10~15个功能点。

这些应用潜在的平均缺陷个数为4.5个/功能点,然而缺陷去除效率只有87%。结果,

在软件第一次交付用户的时候,软件中大约还存在 1400 个缺陷。在这 1400 个缺陷中,约有 20% 的缺陷,或者 280 个缺陷,会导致用户使用该软件的时候出现相当严重的问题。

假如从定制设计和定制编码转变成基于认证的可重用组件的构建,那么可以预见,到时候我们的生产率将达到每人每月 45 ~ 50 个功能点。开发时间将会缩短 1 年左右,过去开发一个应用可能需要 2.5 年,现在或许只要 1.5 年就可以完成。

除此之外,潜在缺陷个数将只有 1.25 个/功能点,并且缺陷去除效率将达到 98% 左右。最终,在交付的时候,潜在缺陷只有 60 个左右。在这 60 个缺陷中,只有 10% 的缺陷会导致严重问题,因此在版本发布后,用户可能会碰到 6 个严重缺陷。

这些在质量上的改进当然有利于客户支持和维护,同时也对初始的开发很有利。

由于设计的例子工具是用来捕捉历史数据的,并且还创建了一个 ISBSG 基准的超集,因此,很显然,工具的开发将包括生产率、进度时间表、员工以及质量基准等。实际上,我们可以这样来设想,即每一个主要的软件应用都会包括这些基准数据,并且它们会定期地被添加到 ISBSG 的数据集中。然而,由于竞争的态势、机密军事安全以及其他一些更重要的因素,某些应用的基准数据也许不适合公开。

这将是一件很有趣的事情,那就是推测完全通过可重用材料来开发一个新应用需要什么。首先,我们需要一个专家系统,它将用来分析代码和大量现有遗留应用的结构。这种分析的理念是,通过检查代码来检查软件的结构和体系,然后使用模式匹配来组合最优的设计模式。

100% 开发的另一个标准就能够获得所有可重用代码、可重用测试用例、可重用用户文档、可重用帮助文档以及其他一些可交付成果的主要来源。并不是所有的这些资料都有单一的来源,因此一个动态的可持续更新的目录也是必需的,并且目录能链接到可重用资料的主要来源。

当组件由多个公司提供,并且使用多种语言和方法来创建时,毫无疑问,可重用组件之间的接口需要严格的定义,并且如果可行,需要标准化大规模的重用组件。

由于质量和安全是核心的问题,因此选择的代码段要么经过高标准的认证,要么经过非常严格的质量审查,审查的过程包括:静态分析、动态分析、安全分析以及可用性分析。

假设所有的标准都已经存在,那么其结果肯定是令人兴奋不已的。一个拥有 2500 个功能点的应用程序,它的生产率可能突破 100 个功能点/月,而开发时间则可能缩短到三至六个月。

除此之外,潜在缺陷个数将不超过 1 个/功能点,而缺陷去除效率则可能达到 99%。一个拥有 2500 个功能点的应用倘若能达到这种水平,那么它在交付的时候,大约只会有 25 个缺陷;而在这 25 个缺陷中,可能只有 10% 的缺陷会导致严重的问题。因此,在交付的时候,大约只有 3 个严重的缺陷会出现。

在 2049 年,不太可能出现自动开发复杂应用的工具,但是我们至少可以设想将需要的技术。我们甚至可以设想一种机器化的软件装配生产线,并且智能工具和专家系统可以完成人类 90% 的任务。

3.5 用户文档

在2009年,客户支持和用户文档是软件应用中两个较为薄弱的环节,并且满意程度经常介于“可接受”与“不可接受”之间。例如苹果、IBM以及联想等少量的公司偶尔达到“好”的水平,但也并不是经常达到。

由可重用组件构建的应用会将帮助文档和用户信息作为安装包的一部分,因此第一步就是为新应用规划可重用材料,并且整合所有文档。然而,对于拥有数十或数百个功能的应用,特定功能的文档并没有任何形式的详细信息,因此还必须为少数的功能创建文档。

对于用户文档和帮助文档,下一步就是派遣一个智能工具或者虚拟化身,去核对所有客户手册、第三方用户指南以及帮助文档中的用户评论,就像在网上讨论一样。显然,在论坛或者讨论组里面,有关这些主题的好评或者投诉都会很多,因此需要一个智能工具来收集并整合这些信息,以获得一个全面的了解。如亚马逊一样,也会分析第三方的书籍评论。

一旦智能工具收集信息完毕,通过使用自动化工具(如FOG和Fleisch指标)以及来自作家和作者的评论,智能工具将分析图书和文本的采样,这些采样从用户那里获得,具有最高最有利的评论。

这项工作的主要目的就是找出图书和用户信息的结构和模式,并且让现场用户提供类似应用的最佳信息。一旦优秀的文档被确定了,那么由于该著作已经获得了类似应用中最佳的评论,因此可以将获取用户信息的任务交给作者。

如果这些作者不愿意提供用户信息,那么至少他们的书可以提供给某些作者,这些作者乐意提供用户信息,并且也乐意创建用户指南。这样做的目的是建立一种成功可靠的模式,这种模式适合所有出版物。请注意,我们并无意侵犯版权。信息的整体结构和顺序是重要的。

许多年前,IBM为他们自己的用户指南做过这类分析。用户的评估报告分析完之后,所有的IBM技术作家都会收到一箱书和指南,这些书和指南已经获得了最高的评分。

其他类型的教程材料包括DVD、网络研讨会、操作系统、电话交换系统、武器系统,也可能是对非常大并且复杂的应用进行的现场指导,例如ERP软件等。除非这种材料能在网上获得,否则通过智能工具将很难分析。因此,在编写培训材料时,人们的洞察力可能仍然会发挥很大的作用。

由于应用会销往很多国家,因此可以用自动化翻译工具作为一个出发点,把文档和培训材料翻译成几种不同的语言。希望在2049年,自动化翻译工具能够翻译出更加通顺、更加地道的文档。然而,最终还是需要人来编辑。

由于这样的工具具有全球市场,因此可以预见,我们会经常将文档翻译成汉语、日语、俄语、德语、法语、韩语、西班牙语、葡萄牙语以及阿拉伯语版本。在某些情况下,也许会被翻译成波兰语、丹麦语、挪威语、瑞典语、立陶宛语或者其他语言。

3.6 客户支持

目前,客户支持比用户信息更糟。客户支持的主要问题包括如下内容,但并不限于以

下内容:

1. 当客户打电话要获得帮助时, 往往要等很长的时间。
2. 对于失聪或者有听力障碍的客户, 电话支持很有限。
3. 一线客服人员缺乏培训, 许多问题都解决不了。
4. 由于工作时间是有限的, 因此客户支持的时间也很有限。
5. 通过发送电子邮件来获取帮助, 但是客服回复得太慢。
6. 在供应商的软件中, 客户支持收费, 甚至报告错误也要收费。
7. 对经常报告的错误和缺陷缺乏分析。
8. 对“常见问题解答”和回应缺乏分析。

由于软件在发布的时候, 就带有很多严重的错误或缺陷, 因此在应用第一年使用的时候, 大约有 75% 的客户服务电话是针对错误和问题的。当软件开发使用认证的可重用组件, 并且新开发的目标是零缺陷的水平, 那么到了 2049 年的某一天, 与 2009 年相比, 错误相关的投诉电话数量将至少减少 65%。这种方式将有助于电话和电子邮件的响应速度。

另外一个问题是, 对于听力失聪和有听力障碍的客户, 我们缺乏对他们的关爱。针对这个问题, 我们需要对软件供应商做大量的工作。要使语音自动地翻译到文本, 我们应该可以使用类似 Dragon Naturally Speaking (一种语音识别工具) 或者其他语音翻译工具, 然而, 我们有希望在 2049 年看到新型的语音自动翻译工具, 该工具在速度和准确率上都有很大的突破。

虽然, TTY (TeleTYpewriter, 电传打字机) 设备和电话公司也许应该为听力失聪和听力有障碍的人提供帮助, 但是, 这样做并不方便处理软件故障报告和客户支持。即使供应商支持电话答复, 但是由于还需要涉及技术术语, 因此用户长时间的等待就在所难免了, 所以这类支持最终将变得尴尬不已。

理想情况下, 手机和固定电话上可能有一个特殊的组合键, 这个组合键供失聪或者听力有障碍的人使用。当失聪或听力有障碍的人点击该组合键的时候, 语音也许应该自动地翻译到屏幕上, 像这样的功能应该由供应商提供, 或者甚至由手机制造商提供。

最重要的是, 有数以百万计的计算机用户是失聪或者听力有障碍的, 他们很需要用户指南和帮助文档, 再加上软件的质量还相当低劣, 这就使得针对聋哑客户的软件客户支持变得相当困难了。

其他形式的身体残疾可能也需要专门的辅助工具, 如失明或者丧失肢体的残疾人。

对于一些常见的错误和问题, 数以百计或者数以千计的客户可能会经常碰到, 因此所有的错误报告需要一个有效的分类, 以使它们能够进入一个存储库, 并且通过专家系统来分析常见的原因和症状。这些高频率的问题需要交给客户支持组织中的每个人。由于错误或问题都是固定的, 并且临时的解决方案是不断变化的, 因此这些解决方案就需要实时地提供给客户支持人员。

有些供应商会收取客户支持费用。供应商对此进行收费的主要原因是想减少来电的数量, 从而减少客户支持人员的数量。让客户报告错误以帮助为客户修复错误本来是一件相当好的事, 但是倘若以此来向用户收费, 那么这项政策就是相当地迂腐了。公司这样做,

经常会使一些用户很恼火，而最终选择了其他的供应商。当然，收费是很愚蠢的办法了，我们有更为行之有效的解决方案，那就是更好的质量控制。

所有收到的问题报告似乎说明了一个问题，那就是软件中真的存在错误，因此部分软件供应商应该立即采取如下行动：

1. 需要使用标准分类工具来分析错误的症状。
2. 通过静态或者动态分析法一次性完成错误分析。
3. 应该不断地定位错误在应用程序中的位置。
4. 应该立即将错误通知给变更团队。
5. 倘若客户多次报告相同的错误，那么就需要对该错误加以警惕。
6. 应该尽快给客户修复错误。
7. 如果错误是来自外部的可重用代码，那么应该及时通知。
8. 严重级别和其他议题应该包含在月度缺陷报告中。

一些大型公司有相当复杂的缺陷报告工具，如 IBM，这些工具能够分析错误，并分类错误症状，也能将错误及时地告知给合适的变更团队，同时也能及时地更新缺陷和质量数据。

顺便说一下，由于这里所列举的工具具有质量和缺陷评估功能，因此该工具当然可以用来递归地评估其自身的缺陷级别。这样就提出了一个推论点，这就是，我们应该使用能改进质量的开发方法，如 TSP 和 PSP、静态分析和审查等。

构建一个客户支持的专家系统在技术上是可行的，该专家系统应该具有以下几个功能：

①语音识别；②语音到文本的翻译；③人工智能引擎，该人工智能引擎能够和客户对话，倾听他们的问题，并且根据客户的问题来匹配对应的报告，同时也要提供状态给客户，而对于独特的或者特殊的情形，智能引擎应该能够将客户的问题转给一个专家，以获得更多的咨询和支持。

事实上，如果专家的缺陷分析报告和先前的客户电话混合在一起，那么人工智能引擎很可能会超越人工客户的支持。

由于这类专家系统不依赖于人工专家来应答初始来电，因此这个系统可以至少降低 10 分钟的客户等待时间。在 2009 年，这样做具有典型的价值，也许电话铃只需要响三声，或者少于 3 秒钟。

将高质量的可重用材料与专家系统的支持（能够分析软件缺陷）相结合能够显著地改善客户支持。

3.7 部署和客户培训

应用程序，如本例中使用的评估工具，一般通过如下四种方式来部署：

1. 通过 CD 或者 DVD 发行。
2. 用户在网站上下载，然后安装。
3. 应用不需要安装，可以直接在 Web 服务器上运行，如 SaaS。

4. 通过供应商或者供应商代理来安装应用。

2009 年,这四种方法的分布正在悄然变化。四种方法的相对比例为:通过 CD 安装的约为 60%,通过下载安装的约为 25%,通过供应商提供进行安装的约为 10%,而直接在 Web 上使用的大约有 5%。

如果按照这种趋势继续发展,那么到了 2049 年,四种方法的分布将会面目一新:直接在 Web 上使用的估计有 40%,通过下载安装的估计有 25%,通过 CD 安装的估计有 10%,而通过供应商来进行安装的约有 15%。

供应商安装一般适应于大型或者专业化的应用,例如 ERP、电话交换系统、机器人制造、进度控制、远程控制、医疗设备、武器系统等,这些设备在安装的过程中都需要大量的定制。

尽管一些应用足够简单,客户使用的时候只需要少量的培训即可,但是大部分的应用都是相当复杂的,并且也较难入门。因此,对于大型软件来说,辅助的教程信息和培训课程是必须的。这种培训可以由供应商来提供,但是一个大型的第三方会推销由其他公司(如图书出版社和专业教育机构)提供的书和培训材料。

由于现场培训的成本比较高,因此我们可以预测,在 2049 年左右,大多数的培训将使用预先录制的网络研讨会、DVD 或者其他的方法,这样做可以让培训材料使用多次,同时也方便安排客户进行培训。

然而,我们也可以这样设想,那就是专家系统和虚拟化身可以在虚拟环境中操作。这样的虚拟化身也许会被当成现场的导师,因为这个虚拟化身甚至能够回答学生们的问题,并且还能够与学生进行互动,但在现实中,这些虚拟化身可能通过人工智能来构造。

由于生产、分发纸质书籍和手册代价高昂,因此,我们可以预计,2049 年左右,几乎所有的教材都可以在网上找到,或者也可以在便携设备上找到,如电子书阅读器、手机或者手持设备等。纸质版可能会按需生产,但是不管怎么样,到了 2049 年,纸质版将会比现在少很多。

3.8 软件维护和功能增强

虽然软件应用的平均生命周期都在 10 年以上,但是软件的开发过程与一个真正的工程学科相比显然有些微不足道。软件一旦初步开发和部署,软件应用的整个生命周期就应该包括维护(缺陷修复)和功能增强(新功能)。

在这里,我们讨论软件的成本评估领域。COCOMO 第一次与公众见面是在 1981 年,Price-S 甚至更早一些,许多评估工具第一次进入市场也都在 1980 年的 6 月左右。实际上,对大型应用来说,目前我们都无法知晓它们最大的期望年限,因为许多大型应用目前仍在运行中,例如航空管制等少量的应用,也许最终会持续服务 50 年以上。

顺便要说的,软件在第一次部署之后,它的规模将会以 8%/ 年的速度增长。因此,该软件在使用 20 到 30 年之后,它的规模将会是过去的 2 倍。不幸的是,这种增长伴随着

循环复杂度^①和基本复杂度^②的严重飙升，最终，将使得维护成本变得非常昂贵，或者在变更期间，使得次生缺陷注入一直不断增加。

要想延长老化遗留应用的使用寿命，那么这些应用在使用了5到7年之后，就需要更新一下。更新主要是为了淘汰易错模块、更新应用或者简化代码的复杂性、淘汰安全漏洞，并且甚至可能将代码转换成更加现代的语言，如E语言^③。我们可以从多个供应商那里选择自动化的翻译工具，据说这些工具的效果很好。在应用进行更新的时候，其中一些工具能够预测功能点的总数。这个功能除了对基准很有用外，并且对研究生产率和质量也是大有裨益的。

例如，这里所列举的评估工具，至少在1年之后才会被加入新的功能。当这些工具出现错误的时候，这些新版本也会修复错误的。

由于新的编程语言以每个月1门的速度出现，并且目前已知的编程语言已经有700多种了，因此当新的编程语言出现的时候，任何支持代码评估的工具必须紧跟新语言的步伐。所以，一个智能工具也会不断地在互联网上搜索新语言的描述和公共报告，以及他们在质量和生产率上的影响。

当一个智能工具扫描过去发布的竞争性评估工具时，智能工具也会添加一些新功能。对于任何商业应用来说，了解直接竞争对手的功能集以及他们的产品是相当重要的。

当然，对软件评估来说，要想成为市场的畅销产品，只是针对竞争对手的功能，采取被动的行动是远远不够的。规划一些新颖先进的功能是很有必要的，并且还得确保这些功能不是当前的竞争性评估工具所具有的。

对于这里所讨论的评估工具，一系列有趣的新功能将会在随后的几年里与大家见面。以下列举了一些新功能，但却不限于这些功能：

1. 开发方法的对比（如敏捷开发、RUP、TSP等）。
2. 包括“设计成本”和“员工成本”的评估。
3. 包括挣值^④的评估和跟踪。
4. 评估六西格玛、质量功能展开等的影响。
5. 评估ISO9000以及其他标准的影响。
6. 评估测试人员、QA人员认证的影响。
7. 评估专家与通才的影响。
8. 评估大型团队与小型团队的影响。
9. 评估分布式开发和国际开发的影响。
10. 评估跨国、跨平台应用的影响。

① 循环复杂度是一种代码复杂度的衡量标准，用来衡量一个模块判定结构的复杂程度，数量上表现为独立线性路径条数，循环复杂度大说明程序代码可能质量低且难于测试和维护，根据经验，程序的可能错误和高的循环复杂度有着很大关系。

② 基本复杂度是指由于一个问题的本质不适合简单的求解方式，所以可行的求解方式都有很复杂的情形。

③ E语言，即易语言，是一门计算机程序语言，以中文作为程序代码的语言。

④ 挣值（Earned Value，EV），表示实际完成的工作所对应的预算成本，在计划和实际之间建立了一个桥梁。

11. 评估已发布应用中的缺陷对客户的影响。
12. 评估大型 ERP 和 SOA 工程的部署成本。
13. 评估拒绝服务和其他安全攻击的恢复成本。
14. 评估外包项目发生诉讼的可能性。
15. 如果发生违约, 要能够评估诉讼成本。
16. 评估专利授权成本。
17. 如果发生专利侵权, 要能够评估专利诉讼成本。
18. 评估重大商业软件缺陷可能带来的一系列损失。
19. 评估应用中的严重错误可能带来的诉讼几率。
20. 项目历史与成本核算包的整合。

很明显, 当应用几乎全部由可重用组件构成时, 软件应用的维护将会比 2009 年复杂得多, 如本节所讨论的应用。功能和代码也许来自十几个供应商, 并且也可能来自几个内部应用。

不管什么时候, 与应用相关的错误被报告出来时, 倘若其他应用使用了相同的可重用组件, 那么该应用可能也会出现相同的错误。因此, 拥有应用中每个功能的确切来源信息是很重要的。当出现错误的时候, 有必要通知功能的源头。如果错误来源于一个已知的内部应用, 有必要通知应用的拥有者以及应用的维护团队。

由于实例应用是跨平台运行的, 因此在一个平台上所出现的缺陷也许会出现在其他平台的版本上。不管什么时候, 当任何版本出现一个重要的错误时, 就需要使用静态和动态分析工具做一个关键类型的分析。很明显, 对于所有的版本, 如果错误似乎有更广泛的影响, 那么我们就必须警告变更团队了。

当然, 只有通过相当复杂的错误分析, 我们才能确定是具体哪个功能出了问题。在 2009 年, 这种类型的分析是由维护编程人员来完成的, 但到了 2049 年, 静态和动态分析工具应该能够更快更准确地定位错误。

到 2049 年时, 维护或缺陷修复应该可以使用强大的工作台, 这个工作台整合了缺陷报告和路由、自动化静态和动态分析、连接测试库和测试用例、测试覆盖率分析器以及复杂性分析工具等, 也许还包括自动化测试用例生成器以及更加专业化的工具(如代码重构工具和语言翻译器)。

由于功能点指标是基准的标准做法, 因此毫无疑问, 对遗留应用来说, 维护工作台也将自动生成功能点计数, 并且改进也足够大, 能够改变功能点总数。

从以往来看, 以功能点作为计量单位, 软件应用的规模会在原始的基础上以每年 8% 的速度增长。我们没有理由认为 2049 年的增长速度会比 2009 年的慢, 但恰恰相反, 我们有理由相信增长速度也许会更快。

首先, 使用智能工具将会快速确定特定的功能, 使用标准可重用组件的开发速度是如此迅速, 因而到了 2049 年, 确定一个有用的功能, 然后将其添加到应用中的时间也会很短, 可能会少于 6 个月, 而 2009 年, 则需要 18 个月。

2009 年, 当软件应用超过了使用年限时, 人们会逐渐废弃原始的需求和设计材料。到

了2049年，只要应用一旦投入使用，智能工具和专家系统相结合，将会使应用紧跟时代的步伐。相同类型的专家系统可以用来挖掘业务规则和算法，因此可以持续使用，并且可以确保软件和它们的支持材料经常具有同一级别的完整性。

这样就提出了一个观点，那就是生产率和质量的基准也许最终会有30年的历史，也可能甚至是50多年。因此，提交数据到基准库（如ISBSG）将会是一个持续的活动，而非一个一次性的事件。

3.9 软件外包

外包公司主要集中在美国、中国、印度、俄罗斯等几个国家。我们不仅需要评估外包公司，而且还要评估更大的商业问题，如膨胀率、政府的稳定性以及知识产权保护等。在当今世界金融诈骗风行的情况下，在选择外包公司的时候需要注意公司健全的财务状况，如印度的Satyam咨询所展示的财务异常。

目前，外包公司的潜在客户谎称自己有很多成果，然而却鲜有真实的历史资料以支撑其言论。笔者曾经在十几个诉讼中作为专家证人，这些诉讼主要是关于外包商违反合同的，在这些诉讼中，笔者发现，供应商在市场上的吹嘘与项目的实际开发方式相差甚远。他们营销声明中声称会将最佳实践贯彻始终，但事实上大多数真正的实践是相当糟糕的，如缺乏评估、掩耳盗铃式的进度报告、缺乏质量控制、糟糕的变更管理以及其他大量的失败。

到2049年，智能工具和专家系统相结合，应该能够提供严谨扎实的商业洞察力，以寻找合适的外包合作伙伴。外包的经营决策分为两部分：（1）对于一个具体的公司来说，外包是否是一种正确的策略；（2）如果外包是一种正确的策略，公司应该选择一个能干的外包供应商。

第一步就是考虑外包是否是一个合适的策略，也就是说要评估你当前软件的效益和战略方向。

由于软件运营变得越来越大，越来越昂贵，并且也越来越广泛，因此许多大型企业的高管都在问一个基本的问题，那就是，软件是否应该成为我们的核心业务的一部分？

这并不是一个容易回答的问题，本章将探究该问题的答案。在以下条件下，你可能想使软件成为核心业务操作的组件部分：

1. 你所卖的软件依赖于你的专有软件。
2. 当前，你的软件给你的公司带来了强悍的竞争优势。
3. 你们公司的软件开发和维护效率远胜过你的竞争对手。

如果外包与你的核心业务具有如下关系，那么你也许应该好好考虑一下软件外包：

1. 软件主要是用于企业运营的，而不是作为一个产品。
2. 与你的竞争对手相比，你的软件没有特别的优势。
3. 你的开发和维护效率微不足道。

在过去的数年里，信息技术架构库（ITIL）和面向服务架构（SOA）已经出现。这些方法强调软件的商业价值，并且促使我们考虑软件作为一种有用的服务，而非一种昂贵的企

业奢侈品。

不管软件是属于企业经营的一个组成部分，还是属于外包的一个组成部分，初步的注意事项涉及的主题包含以下 20 点：

1. 从你目前的软件中，你是否获得了重要的竞争优势？
2. 在你目前的软件中，是否包含商业机密或者有价值的专有数据？
3. 贵公司的产品是否依赖于专有软件？
4. 你们目前的软件从以下哪些商业功能中受益？
 - A. 企业管理
 - B. 财务
 - C. 制造和分销
 - D. 营销及市场推广
 - E. 客户支持
 - F. 人力资源
5. 贵公司目前拥有多少款软件？
6. 有多少新款软件是贵公司在下一个 5 年里所需要的？
7. 你的软件中，有多少是老化的旧系统？
8. 在你们老化的旧系统中，有多少是与 ITIL 兼容的？
9. 在你们老化的旧系统中，有多少适用 SOA？
10. 你们的软件开发效率是否胜过竞争对手？
11. 你们的软件维护是否比竞争对手更高效？
12. 你是否准备好推广软件相关的产品，以超越竞争对手？
13. 你的软件的质量水平是否强过竞争对手？
14. 你是否可以使用大量可重用构件？
15. 目前拥有多少软件员工？
16. 在未来五年里，将会雇用多少软件员工？
17. 在你的公司里，有多少软件使用者？
18. 在未来五年里，会有多少软件使用者？
19. 你是否考虑过企业的软件包，如 SAP 或 Oracle？
20. 你是否发现由于人员短缺，你很难雇到新员工？

由于公司不同，因此回答的模式可能有很大的不同，但是回答主要集中在以下几种可能性：

- A. 如果你的公司在你的领域里，是一个软件巨头，那么你可能就不会考虑外包。
- B. 相反，如果你的公司在软件领域中，落后于主要竞争对手，那么你应该立即采取行动，使用外包。

在另外两种情形下，外包的利弊并存：

- C. 在你的行业里，你的软件操作貌似处于中等水平，在许多方面，既不算太好也不算太差，马马虎虎。在这种情况下，如果你选择了一个好的外包合作伙伴，那么外包可能会

为你减少支出，或者至少给未来带来稳定的软件支出。

D. 另外一种含糊不清的外包情形是这样的：由于长期以来，在你的行业里或你的公司里，你一直缺乏软件方面的数据。因此，你的软件到底是好于竞争对手还是差于竞争对手，你一无所知。

在这种情况下，无知是最危险的。如果你没有通过一种有效的途径知道你的软件所处的水平，那么可以肯定地说，你的公司绝对不是行业里面的巨头，并且你的软件性能甚至连中等水平都不如。当然，可能会更糟，说出这样刺耳的言论是因为，所有真正好的精英软件组织都有质量和生产率度量程序，因此他们对自己的软件所处的水平了如指掌。

你的公司也许应该将内部最近的软件项目实例与开源的行业基准（如 ISBSG）相比较。

一旦一个公司决定使用外包，并且将其作为一种合适的商业策略，那么第二步就是寻找真正能干的外包合作伙伴。所有的外包公司都声称自己能够胜任，并且许多外包公司也真的是名副其实，然而也有一些则是名不符实，夸夸其谈。由于外包是一项长期的策略，因此公司还得从长计议，当选择外包合作伙伴的时候，需要进行严格的调查研究。

你可以和你的员工一起去评估潜在的外包合作伙伴，也可以找一个或多个精通外包领域的管理顾问。不管你通过哪种方式，第一步都是让智能工具获得所有外包公司的信息，当然这些外包公司的业务要与你的业务需求匹配，如 CAI、EDS、IBM、Lockheed、Tata、Satyam 等。

智能工具获取的信息包括：财务数据（如果是上市公司的话）、过去或现在由客户提起的诉讼、由美国证券交易委员会或联邦政府监管机构发起的对公司的调查以及展示生产率和质量结果的基准。

2009 年，关于外包的一个基本的决定是首选本地的外包合作伙伴还是国际的外包合作伙伴。国际的外包公司主要来自中国、印度、俄罗斯等，并且有时他们会提供较有吸引力的方案，使得成本在短期内降低。然而和国际外包合作伙伴进行交流就显得比国内合作伙伴麻烦得多，并且还得评估其他一些问题。

在中国、印度、俄罗斯，最近的经济形势表明已经提高了通货膨胀率。与日元、英镑相比，美元正在贬值，这促使美国成了一个首选的外包国家，例如，由于艾奥瓦州有良好的商业环境和低廉的劳动力成本，因此 IBM 正打算在艾奥瓦州的迪比克市创建一个新的大型外包中心。

美国的外包成本已经比一些主要的贸易伙伴低廉了，如日本、德国、法国等。如果这种形势持续，并且如果美国进入经济衰退期，那么在全球的外包市场，美国最终将具有相当强悍的竞争优势。

然而，到了 2049 年，在全球的外包行业中，一批完全不同的玩家可能参与到全球的外包中，届时也许会出现百花齐放的情形，例如当在写本书的时候，越南正在开发相当快速的软件方法，并且软件专业技术也正在墨西哥、巴西以及其他一些南美洲国家扩展。

假如，中东地区出现持久的和平，那么到了 2049 年，伊拉克、伊朗、叙利亚和黎巴嫩也许会成为全球技术市场中重量级的玩家，斯里兰卡、孟加拉等十几个国家也同样可能。

到了 2049 年，你应该可以通过智能工具获得如下信息：每个国家的通货膨胀率、知识

产权保护法、外包公司的数量、软件工程人员、软件工程学校和毕业生、当地的税收结构、外包公司的周转率^①以及其他一些信息，以帮助你确定最佳的长期合作伙伴。

如果你正在考虑一个国际性的外包伙伴，那么你的评估中应该包括如下一些因素：（1）针对你们公司所使用的这种类型的软件，候选合作伙伴所具有的专业知识；（2）在你的位置和外包的位置之间交流，卫星的适用性或宽带的可靠性；（3）外包供应商所在国家的版权、专利以及知识产权保护；（4）政治动乱或干扰跨国信息流动的可能性；（5）外包供应商的基本稳定性和经济的健全性，以及供应商可能会遭遇的严重的经济衰退。

本地外包公司经常在一定程度上会降低成本或稳定成本，并且也相当方便交流。当然，还有一个较为敏感的话题，那就是外包公司会成为你当前软件人才未来的雇主。本地外包公司也许会做一些安排，让你部分或全部的雇员变成他们的雇员。

一个值得注意的外包问题就是，外包供应商对某一个具体的行业非常精通，如银行、安全、电信以及其他一些方面（可能提供大量可重用材料）。由于可重用技术能给软件带来最佳的全局效率，因此可重用是一些外包供应商能够精简成本的一种绝佳途径。

有 10 种可重用的软件构件是有价值的，并且其中一些外包供应商的可重用材料也许也来自以下 10 种类别：可重用架构、规划、评估、需求、设计、源代码、数据、人机交互、用户文档以及测试材料。

在评估潜在的外包合作伙伴（国内或国际的）时，需要考虑以下普遍的问题：

- ☐ 针对你的公司所使用的这种类型的软件，外包供应商在你的行业里所具有的专业知识。如果外包供应商也为你的直接竞争对手提供服务，那么就需要确保你的外包有足够的机密。
- ☐ 目前客户使用外包供应商服务的满意程度。你不妨联系几个客户，并获得他们的亲身经历。对于那些和外包已经有两三年合同，并且迄今为止一直对外包有很好满意度的客户，我们有必要和他们进行深入交流。一个智能工具也许能够定位这种公司，或者你可以要求供应商提供客户的名单，并且要求只提供那些和供应商有愉快合作的客户。
- ☐ 在外包公司和客户之间的诉讼（活跃的或者最近的）。在处理外包供应商的时候，尽管活跃的诉讼不是一个精彩的表演，但是如果存在这种情形，那么它确实是我们需要考虑的一个因素。
- ☐ 从生产率、质量、可重用以及其他量化因素（使用如 ISBSG 提供的标准基准）的角度来看，和行业标准相比，供应商所提供的软件的性能如何。对于这种类型的分析，功能点指标已经在全世界范围内被广泛使用，并且远胜过其他任何指标，你应该要求外包供应商具有全面的生产率和质量度量，并且使用功能点作为他们的主要度量指标。如果外包供应商没有任何关于生产率和质量方面的数据，那么我们得注意一下了。你也应该要求某种类型的能力证明，例如要求外包供应商在 SEI 的 CMMI 3 以上。

^① 周转率也称“换手率”，指在一段时间内将外包转手买卖的频率。

□ 供应商所使用的项目管理工具类型。项目管理是软件行业中的一个薄弱环节，并且管理者倾向于使用一系列的项目管理工具，包括成本评估工具、质量评估工具、软件规划工具、软件跟踪工具、项目办公室工具、风险管理工具以及其他几种工具。如果候选的外包供应商没有量化评估或度量能力，那么他们的软件性能是不可能强于你们自己的软件的。

这5个主题只是冰山一角而已。在供应商的评审中所涉及的一些主题有：（1）供应商所使用的项目管理工具和方法；（2）供应商所使用的软件工程工具和方法；（3）供应商所使用的质量保证方法类型；（4）可重用材料的实用性；（5）供应商所使用的配置控制和维护途径；（6）供应商的管理和技术人员的跳槽率和流失率；（7）对于成本控制、进度控制、质量控制等，供应商所使用的基本度量和指标。

ISBSG已经收集了5000多个软件项目的数据。收集新数据的速度大约在500个项目/年。这些数据具有商业用途，并且可以提供有用的商业信息，以让你确定你们公司的成本和生产率是好于平均水平还是差于平均水平。

在签订一份长期的合同之前，客户应该要求潜在的供应商根据以下主题提供量化数据：

1. 以前通过功能点和代码行所构建的应用的规模。
2. 缺陷去除水平（平均、最大、最小）。
3. 任何认证，如CMMI级别。
4. 员工每年的流动率。
5. 任何和外包供应商相关的诉讼。
6. 任何和外包供应商相关的政府调查。
7. 参考其他客户。
8. 外包供应商所使用的质量控制方法。
9. 外包供应商所使用的安全控制方法。
10. 外包供应商所使用的进度跟踪方法。
11. 外包供应商所使用的成本跟踪方法。
12. 外包供应商所使用的认证可重用材料。

自动化的软件评估工具非常实用，如本章所列举的工具，对于相同的项目，允许同时进行评估。一个版本使用当前的内部培训方法，以展示成本和进度报告；第二个版本使用它们的独特方法以及可重用材料，以知道外包供应商是如何构建相同产品的。

笔者曾经是诉讼中的专家证人，在十几个外包供应商和他们不满意的客户之间的诉讼中，笔者发现了几个在外包合同中需要定义清楚的关键议题：

1. 对于协议中所包含的应用程序，使供应商达到预期的学习曲线。假设外包团队成员解决每个功能点大约需要20分钟，那么在加快速度的情况下，两个星期的时间大约可以完成1000个功能点，或6个星期的时间可以完成1万个功能点。

2. 用清晰的语言定义如何处理和发现变化中的需求。所有大于50个功能点的变更需要更新成本和进度评估，并且更新质量评估。不影响功能点总数的需求波动，也需要包含在合同中。

3. 外包供应商所使用的质量控制方法应该被证明是有效的。在外包合同中, 缺陷去除效率达到 95% 以上也将会成为有用的条款。缺陷跟踪率和质量度量也应该是必需的。对于用 C、java 或者其他语言编写的应用, 静态分析也应该是必需的。

4. 在外包合同中, 跟踪和报告开发进度一直是一个薄弱环节。每个项目应该每月被跟踪一次, 并且在开发过程中, 应该向客户报告所有可能影响项目进度、成本或质量的问题。如果发生诉讼, 这些报告将会是发现过程的一部分, 并且供应商将会推翻任何错误或隐瞒这个问题。

5. 在签订合同之前, 合同中应该包含双方共同终止合同的条款, 并且双方都应该了解这些条款。

6. 如果合同中包含延期交付和成本超支需要扣罚金的条款, 那么相应地, 当提前完成, 也应该相应地给予奖励和额外的津贴。然而这些条款都应该与质量和进度挂钩。

在许多管理者看来, 许多外包合同都是模棱两可的。外包合同应该清楚了, 陈述预期的质量结果、处理需求变更的方法以及监控进度的方法。

公司拥有的一些软件也许有相当强的竞争价值, 以至于你可能不想将它外包出去, 或者甚至不想让其他公司知道它存在。在做外包安排之前, 其中一个准备阶段就是调查你们当前主要的系统, 并且对有竞争价值的软件财产做好保密工作。

对当前的系统进行调查, 对你的公司会有很多很好的帮助, 并且, 如果你根本没有考虑外包安排, 你也许想接受这种调查。当前和计划中的软件财产的调查应该涉及如下重要议题。

到了 2049 年, 在软件领域, 智能工具和自动化的业务规则提取工具应该能够提供巨大的帮助。实际上, 大多数的业务规则、算法和专业数据应该从遗留应用中提取, 并且通过使用 AI 工具和智能工具, 投入可扩展和可访问的形式。

- ❑ 识别系统和程序。这种程序有很高的竞争价值, 或者使用专利或商业机密的算法。这种系统也许被排斥在很普遍的外包安排之外。如果它们被包含在合同中, 那么针对保密因素, 应该协商具体的保障措施。也应该注意, 当使用国际合同时, 保留专有或有竞争性的软件和数据是相当微妙的。请确保本地的专利、版权和知识产权可以有力地保护你们的敏感材料, 你或许需要几个国家的律师。
- ❑ 使用你们的软件应用对数据库和文件进行分析, 并且在外包安排中, 有对应的开发策略可以保留机密数据。如果你的数据库中包含有价值 and 专有的信息, 如商业机密、竞争对手、专有客户、员工考评、待定或活跃的诉讼等, 那么在任何外包安排中, 你需要确保这些数据被细心呵护。
- ❑ 量化你关键系统的用户数量, 以及他们对当前关键应用的满意度和不满意度。特别地, 你想要确定任何紧急的改进, 也需要被传递给一个外包供应商。
- ❑ 在外包合同中, 应该包含你目前投资规模的定量分析。通常情况下, 量化应该基于功能点指标, 并且包含当前所有系统和应用的规模 (功能点指标度量), 除此之外, 外包供应商将承担维护的责任。
- ❑ 在外包安排中, 应该包含对未来部署和评估的分析, 或者完成了软件项目的一部分,

后面的由外包供应商来接着开发。你希望了解自己的生产率和质量，然后将你预期的结果和外包供应商所承诺的结果相比较。在这里，使用功能点度量是当前外包合同中最常见的最佳选择。

由于外包合同可能会持续很多年，并且花费数百万美元，因此在签订一个外包合同之前，我们有必要三思而后行，切莫操之过急。

截至 2009 年，对于典型的外包合同会持续多久，有多少最终是合作愉快的，有多少最终是不欢而散的，并且有多少最终会通过法庭裁决的，目前并没有一个全面的调查。然而，笔者曾经在许多诉讼中作为专家证人，在这些诉讼中，笔者发现 75% 的外包合同最终合作愉快，大约 15% 的外包合同最终出现了麻烦，而大约 10% 的外包合同最终出现了诉讼。

通过使用智能工具和专家系统来增强调查，有希望到 2049 年，90% 以上的外包合同最终双方都满意，而诉讼的比例将降到 1% 以下。

由于全球经济衰退的延长和深化，外包可能会以不可预知的方式受到影响。不利的一面是，一些外包公司和他们的客户（或两者）可能走向破产。有利的一面是，对于收入和盈利能力下降的公司来说，符合成本效益的外包是一种省钱之道。

在外包协议中应该加入一个新的重要主题，这就是，从 2009 年开始，假如一方或双方的合作伙伴走向破产，那么合同和软件应该怎么办。

3.10 软件包评估和收购

在 2009 年，购买或租赁一个软件包是一件相当麻烦的事情。供应商的索赔往往是夸张的、不切实际的；软件保证和担保正在消亡，实际上，它损害了客户的利益；质量控制的责任在主要的供应商，例如微软对边际控制得不好；并且当客户需要帮助的时候，客户很难获得支持，即使客户获得了支持，也未能使客户满意。当前，存在许多严重的安全漏洞，因此使得黑客有机可乘，如本书第 2 章所探讨的窃取专有数据或拒绝服务攻击等。

尽管存在这么多的问题，然而大型企业每天依然有 50% 以上的软件是来自外部供应商或开源供应商。就像嵌入式软件一样，几乎所有的系统软件都来自供应商，如操作系统和电话交换系统。其他的大型商业软件包包括数据库、信息库和企业资源规划（ERP）等。

2049 年的情形会比 2009 年更好吗？到了 2049 年，希望认证组件的构建能够改善商业软件的质量、安全以及可靠性。鉴于本章前面探讨了一些方法，因此希望客户支持也有所改善。

到了 2049 年，在获得一个软件安装包之前，我们的出发点应该是先使用一个智能工具去扫描互联网，并获得有关如下主题的信息：

1. 根据需要提供相同或类似服务的所有安装包信息。
2. 通过期刊和审查机构来评论所有的安装包。
3. 所有与安装包相关的用户协会。
4. 公共软件供应商的财务信息。
5. 用户对软件供应商的诉讼信息。

6. 政府对软件供应商的调查信息。
7. 通过静态分析工具和其他方法获得的质量结果信息。
8. 安装包中的安全缺陷或漏洞信息。

2009 年, 软件厂商通常会拒绝提供任何量化的数据。除了一些开源的软件包, 应用的规模、生产率、客户报告的错误、静态分析工具的运行结果等信息是不向客户公布的。除了一些小事或者对客户可能造成损害的事(如销售客户信息), 他们拒绝提供任何保证和担保信息。几乎所有的软件保证都包括免责声明, 即错误或安全漏洞所造成的伤害或损害不归软件方。

软件有如此糟糕的保证是因为有一个隐藏但根本的原因, 那就是软件控制需要考虑很多的方面, 如商业、医药、政府和军事行动, 从而软件故障可能会导致更多的问题和开销, 几乎超过了任何其他类型的产品故障。软件缺陷可导致死亡, 如医疗设备出现故障、飞机和火箭发生故障、空中交通事故、武器系统发生故障、生产停工、关键业务数据错误以及其他许多非常严重的问题。如果由于软件的错误, 软件公司应该承担相应的损失或业务亏损, 那么成功的诉讼甚至可以消灭主要的软件供应商。

个人和小公司一般通过零售方式购买套装软件, 并且他们都没有能力去改变软件供应商相当不专业的营销方式。但是, 大公司、军事机构、联邦政府、州政府以及其他一些大型企业却有足够的影响力, 并且应该坚持在软件包开发、保证、担保、安全控制、质量控制及其他相关问题上做出改变。

虽然在购买有重大质量和安全漏洞的软件包时, 智能工具和专家系统可以帮忙将风险降到最低, 但是这可能需要政府的干预, 以提高担保和保证。然而, 良好的保证将是一个相当强大的营销工具, 如果一个像 IBM 一样的主要供应商开始提供有意义的保证, 那么所有的竞争对手将被迫效仿, 不然将失去大部分的业务。

软件供应商最起码应该提供一项服务, 即客户在购买 90 天后, 倘若不满意, 可以全额退款。虽然供应商可能会因此失去少量的钱, 但是如果保修信息刊登在他们的广告和包装上, 他们很可能会有相当多的额外收入。

2049 年, 对于收购大型套装软件(如微软、IBM、SAP、甲骨文等厂商)的大客户来说, 下面的信息是租赁或购买商业软件产品时应考虑的:

1. 应用的规模(功能点和代码行)。
2. 在开发过程中使用的质量控制步骤。
3. 在开发过程中使用的安全控制步骤。
4. 在产品发布之前, 发现的错误和缺陷数量。
5. 客户的产品中所报告的错误和缺陷数量。
6. 不满意的客户对供应商发起的诉讼。
7. 对主要的缺陷修复, 预计客户支持率。
8. 在缺陷报告出来后, 预计缺损修复后的转变。
9. 保证向用户报告缺陷, 不收费。
10. 保证支持客户的电话或 E-mail, 不收费。

11. 在安装后的 90 天之内, 产品包退。

以上大部分的信息理所当然卖方会声称是自己的专利并保密的。然而, 由于主要客户将会知道这些信息, 因此毫无疑问, 可以在签署保密协议的情况下, 给客户提供一些信息。

随着全球经济衰退的深化和延长, 这会对软件行业以及商业供应商平添许多不必要的问题。从 2009 年起, 在主要的软件合同中应该包含一个新的条款, 即倘若供应商或者客户破产, 我们应该怎么处理软件、担保以及维护协议。

3.11 技术选择和技术转型

软件行业自诞生以来, 技术选择和技术转型一直是软件行业中的两大弱点。软件行业在选择开发方法的时候, 很少会基于成功的可靠的经验数据。相反, 软件行业的运作或多或少地像邪教组织, 领导者一般都具有很强的号召力, 他们开发的各种方法会被众人所推崇。系统一旦开发成功, 这些方法就将获得众多的信徒和弟子的支持, 这些人会继续使用并捍卫该开发方法; 然而这些方法往往很少或者根本没有历史数据, 因此至于效果如何, 无人知晓。

当然, 有些方法还是相当有效的, 或者至少对某种规模和某种类型的软件是有效的。有效的开发方法包括 (按英文字母顺序排列) 敏捷开发、代码审查、设计审查、迭代开发、面向对象 (OO)、Rational 统一过程 (RUP) 以及团队软件过程 (TSP)。其他的方法似乎并没有太大作为, 如 CASE、I-CASE、ISO 质量标准, 当然还有传统的瀑布方法。对于一些较新的方法, 目前还没有足够的数据来确定其有效性。这些新方法有 20 多个, 包括极限编程、面向服务架构 (SOA) 等。少数的几个项目要么实际测量生产力, 要么实际测量质量, 因此我们很难判断它的有效性。

如果软件工程作为一门工程学科, 并取得了实质性的进展, 而不是作为一种艺术, 那么测量和实证结果应该比过去更常见。在美国, 有一个非营利性的评估实验室, 其职责是研究什么对软件行业有益, 类似于消费者协会或保险商实验室, 甚至是美国食品和药物管理局 (FDA) 等。

该组织将在受控条件下评估各种方法, 然后报告它们如何很好地在各种情况下使用, 包括各种类型的软件、各种规模的应用以及各个技术领域, 如需求、设计、开发、缺陷去除等。

将各种开发方法的结果与标准基准样例做一个横向比较, 将是一件非常有趣、同时也是非常有用的事情, 这些开发方法包括敏捷开发、净室开发、智能工具开发、迭代开发、面向对象的开发、快速应用开发、Rational 统一过程 (RUP)、团队软件过程 (TSP) 以及各种 ISO 标准等。

假如没有正式的评估实验室, 那么我们也可以使用第二种改善软件选择的途径, 即收集每一个软件项目的生产率和质量方面的可靠基准数据, 并提交给一个非营利性的信息交流中心, 如国际软件基准组织 (ISBSG)。

历史数据和基准需要几年的积累才能使我们获得足够的信息, 以进行统计研究和多元回归分析。然而, 随着时间的推移, 用基准来衡量进度将非常有用, 而在消费者实验室的

评估只针对一个固定的时间点。

即使开发方法已被证明是明显成功的,但是该事实本身并不能保证他人采用或使用该方法也能成功。通常情况下,除非他们已经使用过该方法,否则可能涉及社会因素,大多数人都将选择墨守成规,不愿意放弃固有的方法。

这不单单是一个软件问题,在人类努力探索的每个领域里,创新和新的实践都会遇到这个问题,如医疗实践、军事科学、物理学、地质学以及其他几十个学科。

一些重要著作涉及技术选择和技术转型的问题。虽然这些书和软件没有任何关系,但是它们为软件社区提供了很多值得借鉴的东西。一本书是 Thomas Kuhn 的著作《The Structure of Scientific Revolutions》(科学革命的结构)。另一本书是 Paul Starr (1982 年普利策奖得主)的《The Social Transformation of American Medicine》(美国医学的社会转型)。第三本重要的书是 Leon Festinger 的《The Theory of Cognitive Dissonance》(认知失调理论),这本书涉及意见形成的心理。

技术转型还有另外一个社会问题,那就是一些行政人员和管理人员经常被误导,他们强制参与者使用这些方法。强制采用的方法通常会失败,并导致怨恨。

一个更有效的方法部署途径是将正在使用的方法作为控制实验。我们可以这样理解,即在经过一个适当的试用期(6 星期至 6 个月)后,我们再对该方法进行评估,以决定该拒绝还是采用这种方法。

当这种控制实验使用如正式检查一样的方法时,其结果几乎总是如出一辙:可以采用该技术。

技术选择中另一个棘手的问题是,许多开发方法的使用范围是很狭窄的。一些工作最适合小型应用程序,但是对大型系统是无效的。还有些被设计成大型系统,它们对小项目和小公司(如更高层次的 CMMI)来说,就显得过于烦琐了。有人是这么认为的,即倘若一种方法为一个小样本提供了很好的效果,那么对所有已知规模和类型的软件来说,使用该方法同样可以获得很好的效果,这样的想法显然是错误的。

分派智能工具有价值的方面之一是,它们可能有能力捕获并显示流行开发方法的利弊,如敏捷开发、TSP 以及其他相关主题,如 CMMI、TickIT、ISO 标准等。

到了 2049 年,软件实践都是基于实际数据和经验结果的,这将是很好的,但这绝不是一定的。实际数据的迁移至少需要 15 年的时间,因为数以百计的企业需要建立度量方案,并且通过有效的度量方法来培训从业者。我们需要购买自动化工具,当然,工具的成本也应该是合理的。

另一种影响软件行业的社会学问题是,一些广泛使用的度量方法要么违背了标准的经济学假设,要么定义得含混不清,以至于它们不能用于基准测试和比较研究。“代码行”指标和“平均缺陷成本”指标都违反了经济学原理,并且应该把其看作经济分析上的专业过失。其他指标如“故事点”和“用例点”可能限制了具体项目的实用性,并且不能用于大规模的经济分析。假如这些项目都不使用用户故事点和用例点,那么这些度量方法也不能用来做横向比较。

进行有意义的基准和经济研究,要么开始的时候使用标准的指标收集数据,如 IFPUG

功能点，要么使用自动转换工具，可以将一般的指标（如“代码行”、“故事点”、“COSMIC 功能点”等）转换成标准指标。很明显，无论是项目组合还是整个软件行业的大型经济研究都需要用标准指标表示数据。

2009 年前后，使用离奇的非标准指标的常规做法是软件行业的一个标志，即软件工程还不是一门真正的工程学科。在 2009 年，关于软件，往好里说，它是一门手艺或艺术形式，有时会产生有价值的结果，但往往都以失败告终。

几年前，通过对 IBM 的技术转型进行研究，我们发现只有约 1/3 的应用程序当时使用了最佳实践。这导致 IBM 投入了大量的资源，以改善公司内部的技术转型。

在惠普和 ITT 所进行的类似的研究也透露了这种现象，即缓慢的技术转型以及非常主观的技术收购。这些长期存在的问题需要一部分的社会学家、工业心理学家以及软件工程专业自身进行更大量的研究。

3.12 企业架构和项目组合分析

一旦智能工具、专业的设计工具以及专家维护工作台被广泛地使用了，将开启新的工作形式，并在企业和项目组合层面处理更高水平的软件所有权。

2009 年的今天，企业和政府机构拥有数以千计的软件应用，这些应用花费了多年来开发，并且使用了不同的架构方法、设计方法、开发方法以及编程语言。此外，项目组合中的许多应用程序可能是商业软件包，如 ERP 套件、办公软件以及财务软件等。这些应用程序都是随机进行维护的，并且大部分的应用都含有相当多的潜在错误。有的甚至含有“易错模块”，这些模块都有非常复杂并且非常容易出错的代码段；通过修复错误，新错误的不良修复注入率也许可以达到 50% 了。

派遣相同的智能工具，并使用相同的专家系统对整个项目组合进行全面详细的分析，这样做在商业的着眼点上是很明智的。这项工作的目的是识别目前所有应用的质量和安全隐患，描绘出如何和当前的应用进行交互，并把每一个应用以及它们的功能集放在标准分类和标准功能的地图上，这些功能使用可重用组件以支持开发。

一个需要专家分析和智能工具的附加功能用来确定软件变化的部分，由于政府规章和法律都在不断地变化，如税法的变化、政策的变化以及隐私需求的变化等，因此软件可能也需要不断地更新。无论是州还是联邦的法律法规，几乎每天都有一些变化，因此只有结合智能工具和专家系统，才能知道数以千计应用的项目组合可能需要些什么。

换句话说，将有可能对整个项目组合进行大规模的数据挖掘，并且提取所有的算法以及整个公司或政府机构所使用的业务规则。也可以通过数据挖掘来构建企业的数据字典。由于大的项目组合可以包括 10 000 多个应用程序以及数 10 万个功能点，因此这项工作通过人工完成并不容易，需要通过自动化的方式来做。

毫无疑问，业务规则和算法会有成千上万个。一旦提取完毕，显然要对这些业务规则和算法进行分类，并根据不同的分类方法（如 Zachman 架构方法或其他定义应用类型、功能类型的分类法）组装成有意义的模式。

这种形式的数据挖掘不仅整合了业务规则，合理调整了项目组合管理以及政府协助，而且它也引进了更好更严谨的经济分析、管理、质量控制以及安全控制等。

可以创建一个巨大的数据字典和目录，以显示所有已知的政府法规对公司项目组合中每一个应用的影响。这样的工作显然超过了人工的极限，只有专家系统、AI 工具以及智能工具能够竭尽全力地做到这一点。

实际上，很少有公司知道他们项目组合的规模（以功能点或代码行度量）；很少有公司知道他们在缺陷修复、改进功能以及其他类型工作上的维护成本分析；很少有公司知道他们现有软件当前的质量水平和安全漏洞；很少有公司知道他们的每个应用到底有多少用户在使用，或者他们的每个应用对组织的价值到底有多大。

到了 2049 年，我们可以这样设想，有一套智能工具和专家系统在不断地工作，以查明遗留应用中的缺陷，并注意有质量和安全漏洞的那部分。代理将分散到 50 个不同的地理位置上，然后由各个地理位置上的公司负责开发和维护。然而，所有用这些工具进行分析的结果将在企业的层面上进行合并。

在收集并分析这些数据后，将它储存在一个活跃的资料库中，即添加新应用、更新当前应用或者废弃遗留应用，基本上每天都可以更新资料库。在系统信息库中存储的数据类型有：应用程序的规模（以功能点和 LOC 度量）、缺陷和变更历史、安全状态 and 已知漏洞、用户数、基于标准分类的功能以及与企业、供应商或客户所拥有的应用之间的关系。

当企业的业务需求和企业的软件组合能够可靠地映射，并且通过自动化工具，所有已知的业务规则和业务算法在现有的项目组合上得到了整合时，我们也可以在企业架构和项目组合管理的层面上预想更好的规划。

软件项目组合以及它们包含的数据是多数公司最宝贵的资产，但在开发、替换和维护时，它们也是最麻烦、最容易出错，而且代价高昂的。

很明显，软件需要从一门手艺转换成一门工程学科，手艺是通过手工编码来构建应用程序的；而工程则是通过标准组件来构建高品质和高安全应用程序的。这项工作组合智能工具、专家系统、架构方法以及几种分类才能完成。此外，应该一直使用静态和动态分析的自动化安全分析和质量分析，只有这样才能保持应用程序的安全和可靠。

这种自动化项目组合分析的经营宗旨将包括公司治理、兼并和收购、软件资产应课税值^①的评估、维护计划、知识产权诉讼、违反合同，当然也包括安全和质量的改进。由于最近的经济形势相当低迷，因此每个公司都需要找到降低项目组合维护成本的方法。只有当专家应用可以完整地扫描并分析项目组合时，我们才能实现真正显著的经济体。

当大型企业之间进行兼并和重组时，项目组合分析就显得尤为重要了。合并两家大公司的项目组合和软件组织是一项艰巨的任务，往往会损害双方的合作伙伴。我们需要对双方的项目组合、双方的数据字典、双方的业务规则和算法进行仔细的分析，但是光通过人工很难做到。显然，智能工具和专家系统对尽职调查以及随后的合并将是非常有益的。

在企业架构和项目组合分析的水平上，展示整个企业的软件使用情况和状态，并使用

① 应课税值是指有纳税义务的单位和个人应上交的货币或实物。

图形化来表示将是很有价值的。一个功能类似于我们目前使用的谷歌地球，它可能一开始会在一个很高的层次俯瞰整个企业和项目组合，然后缩小视线，可以看到单独的应用程序、个别业务单位，甚至可能是个别的功能和用户。

谷歌地球和公司项目组合的整体表现之间的主要区别是，项目组合将显示动画和实时的信息。业务信息流通过连续的动画来表示，包括从单元到单元、从公司到供货商、从供应商到客户以及从客户到客户。

另外一点也是很重要的，那就是在美国税收服务中，软件组合属于应税资产。在并购后，会有频繁的税务诉讼，涉及遗留应用的原始开发成本。随着时间的推移，减少每家公司的税务后果，知道项目组合中每个应用程序的规模、原始的开发成本以及连续的维护和功能增强成本，这将是审慎的做法。

3.13 软件学习预览

目前，由于技术转型是一个薄弱环节，因此在 2049 年，一个有趣的话题是，考虑软件专业的人才可能如何学习软件专业知识。

就 2009 年所提供的技术来预测未来，我们可以肯定地说，教育和学习可能与现在截然不同。这个简短的讨论提供了一个假设性的前提，那就是我们在 2049 年进行学习。

假设到了 2049 年，你有兴趣了解那时的软件生产力和质量的基准。

到 2049 年，几乎所有的材料都将在网上提供各种各样的格式。从一种格式转换为另一种格式将是大家习以为常的。毫无疑问，自动从一种语言转换到另一种语言将很方便，如俄语翻译成英语。

到 2049 年，已出版材料的版权和支付将有望得到解决。在理想的情况下，大量材料的文本挖掘将会在数以百万计的文件上建立有用的交叉引用和索引。

首先，到了 2049 年，你的计算机可能会和今天的普通计算机有所不同。也许会有几个屏幕，并且也有独立的处理器。其中一个处理器将是非常安全的，主要是用来处理 Web 访问，而另一个也很安全，不会直接连接到 Web 上，用来处理文字、电子表格、图形以及其他的活动。硬件安全将是两个处理器共有的一个特征。

电脑键盘可能仍然存在，但毫无疑问，语音命令和触摸屏将会普遍使用。由于今天已经存在 3D 影像技术，因此不管以后你是否使用特殊的眼镜，你都有可能以 3D 的形式查看信息。显然，虚拟现实对教学辅助是很有裨益的。

由于在一个固定的地方看书，人没多久就会感觉累，因此屏幕将会是可拆卸的，或者会附带补充屏幕，并且可以像书一样拿起。以后，屏幕最有可能的形式是类似今天亚马逊的 Kindle 或索尼的 PR-505。这些设备的规模、形状和平装书差不多。毫无疑问，2049 年，高分辨率的图形和真彩也将适用于电子书，并且也可能会有动画。声控命令和触摸屏可能也会建立一定的标准了。到了 2049 年，电池将会更加有效，一个手持设备使用 8 到 10 个小时应该是很常见的事情了。

其他变化的技术可能就是修改计算机的物理外观了。例如，当前灵活的平板屏幕，会

在眼镜的镜片上显示。无论计算机的物理形状如何变化,上网以及获取网上的资料仍将是计算机的一个主要功能,当然安全也是一个不容忽视的问题。

到 2049 年,基本上所有的信息都可以在网上获取到,并且你将会有一个私人的图书管理员虚拟化身,它会为你提供你感兴趣的信息。每天,你都会获得实时变化的摘要,当然这些都是你所关心的主题。

输入个人的学习场所,你就可以搜索基准信息了。该地区可能会出现你最喜爱的校园的 3D 影像,并且还有树木、建筑物以及学生和同事的虚拟化身等。

你可能会开始使用语音或关键字查询,例如:“我目前软件的生产率和质量基准。”

你的虚拟化身可能会询问其他的信息,以缩小搜索范围,如,“你想要开发、维护、客户支持、质量或者安全基准方面的信息?”当然,你可能主要关注“开发的生产力基准”。

进一步缩小搜索范围可能就是一个问题了,如,“你想要 Web 应用程序、嵌入式软件、军用软件、商业应用或一些特定形式软件的开发生产力基准?”

你可能会将问题的范围缩小到“嵌入式软件”。这样你的虚拟化身则可能这么回答,“国际软件基准组的嵌入式应用有 5000 个是来自美国的,7500 个来自中国,6000 个来自日本,3500 个来自俄罗斯,以及 12 000 个来自其他国家。还有 5000 个嵌入式基准是来自其他组织的。你想要整体的基准,还是想要两个国家基准的比较?”

你可能会回应说:“我有趣比较美国、中国、日本、印度以及俄罗斯的基准。为了保持一致性,只能使用 ISBSG 基准数据。”

虚拟化身可能会问,“你是否对具体的语言感兴趣,如 E 语言、Java 语言、Objective C 或者所有语言?”在这种情况下,你可能会回答“所有的语言。”

虚拟化身可能会问,“你是否对具体的方法感兴趣,如敏捷开发、团队软件过程或者能力成熟度水平?”你可能会回应,“我想比较敏捷开发和团队软件过程。”

你的虚拟化身可能会说,“每个国家约有 1000 个嵌入式应用使用敏捷方法,约有 2000 个应用使用 TSP 方法。几乎所有的嵌入式应用都在 CMMI 3 级以上。”

在这一点上,你可能会这样说,“为功能点在 1000 和 25 000 之间的嵌入式应用创建嵌入式生产力水平比较图表。比较敏捷开发和 TSP 之间的异同。同时,也展示功能点为 1000、5000 以及 10 000 等三种类型的嵌入式应用的最高生产力水平。”

在几秒钟内,你将会获得一组初始的图表。然后,你也许决定缩小搜寻范围,可以要求最近 10 年的年度趋势,或许包括其他的因素,如查看军用与民用嵌入式应用。

你可能也会问虚拟化身一些日程安排,如即将举行的关于基准方面的网络研讨会和讲座。你可能还要求虚拟化身为你准备过去 6 个月内基准方面的网络研讨会和讲座中的总结要点。

此时,你也可能会要求你的虚拟化身将图的副本发送给在同一领域做研究的同事们。毫无疑问,到 2049 年,所有的专业人士都将链接到一些社会网络,一起处理共同关心的话题。

在 2009 年,已经出现了使用商业服务的社交网络,如 LinkedIn、Plaxo、各种论坛、Wiki 群组等。但是对于大量共享的信息来说,今天的网络显得有点捉襟见肘了。

虽然这种情况是假设的,可能不会发生,但是 2049 年的学习与 2009 年的学习之间有

如下主要区别：

1. 我们的电脑与 2009 年相比，有更好的安全性。
2. AI 虚拟化身或智能工具根据配置文件和个人兴趣，协助处理大量的信息。
3. 与 2009 年相比，文档之间有更好的索引和交叉引用功能。
4. 对于处理数以百万计文档的版权和支付，拥有可行的方法。
5. 积累私人“图书馆”的网上信息，以满足你的个人需求。要使信息发挥更大的作用，智能工具还需在你的整个集合上创建交叉引用和索引。大量的信息都可以在网上获取，多数信息都可以通过像 Kindle 一样的手持设备来获得，也可以从你的电脑、智能手机或其他无线设备获得。
6. 所有符合个人兴趣的网络研讨会、座谈会以及其他交流形式的时间表。这些可以即时看，也可以存储起来供以后观看。你的图书管理员虚拟化身也会以摘要的形式为你提取相关的信息。
7. 有专门的社交网络，用来让同事们交流，并分享软件生产率、安全性、质量等方面的研究成果和数据。
8. 有与虚拟社区相关的社交网络，可以让你和你的同事在虚拟环境中参加在线讨论和会议。
9. 通过利用标准分类的知识，以方便组织数以百万计的文件，这些文件涵盖数千种主题。
10. 开发出相当复杂的过滤器，以便把高价值的信息从低价值的信息中分离出来。例如，倘若文献缺乏生产力方面的度量数据，那么它所拥有的价值很可能比含有度量数据的文献要低。

目前，Web 和互联网上有大量可用的数据和信息。但是，这些数据是混乱的、非结构化的，并且知识的内容也不统一。希望到了 2049 年，能结合标准分类、元数据、虚拟化身以及智能工具，收集任何已知主题的有用信息，过滤掉低价值数据，并将高价值数据浓缩成有意义的集合。

此外，2049 年，各个领域数以百计的同事将通过社交网络联系在一起，这样他们每天都能够分享数据，并迅速研究任何领域最先进的知识。

既然有这么多的信息，那么版权和付款方式必须是安全可靠的。此外，与 2009 年的规范相比，私人数据的采集和在在线文档库的安全性必须是非常强健的。许多信息也许需要加密。硬件的安全方法可能会导致软件的安全方法也扩充。但是，智能工具可以以你的名义来操作，从数十亿的源文件中提取有用的信息。

3.14 尽职调查

虽然经济衰退已经放缓风险资本的投资，并使软件 IPO（上市）几乎处于停滞状态，但兼并和收购的步伐并没有放慢。事实上，一些合并的公司与衰退的经济形势背道而驰，如 Corum 在 2008 年就创造了奇迹。

不管什么时候,尽职调查是必需的,并且始终需要并购和私人投资方面的消息,显然,我们将分派智能工具和专家系统来评估双方的项目组合和应用。

如果公司是中型或大型公司,那么它们各自会拥有 1000 多个应用,并且功能点总数会超过一百万。真正的大型公司能拥有的软件数可能是这个的 10 倍。通过人工来完成整个项目组合的尽职调查实在是太困难了,只有智能工具和专家系统能够处理如此大量的软件。

当合并完成后,就需要软件投资方和软件开发方整合双方的资源,或者至少需要共同经营一些应用。

因此,应该由智能工具和专家系统来检查应用程序的每一个方面,如安全漏洞、潜在缺陷、接口、已有的数据字典、可重用材料等。

在只有一个或两个软件应用的初创企业上进行风险投资,专家对软件质量、安全漏洞以及其他议题进行的分析,将有助于判断投资到底会实现盈利,还是会出现亏本。

正如前面所提到的,软件是一个应纳税的资产。因此,每一个软件应用需要永久地保存应用规模、原始开发成本、维护和改进成本、营销成本以及其他财务数据记录。质量和可靠性数据也应该保存,为可能出现的客户或用户诉讼提供防御性的援助。

在尽职调查活动期间,需要对一些主题进行评估。以下罗列了一些主题,但并不限于以下主题:

1. 软件资产中知识产权保护(专利、商业秘密)。
2. 正在进行的诉讼(如违反合同、税收等)。
3. 以往应用的生产率和质量基准。
4. 软件开发中所使用的质量控制方法。
5. 旧版软件在缺陷和可靠性方面的数据。
6. 旧版软件在客户满意度方面的数据。
7. 在软件应用程序中使用的安全控制方法(如加密等)。
8. 在企业层面所使用的安全控制方法(如防火墙、防病毒等)。
9. 遗留应用中所存在的业务规则、算法等。
10. 企业架构的模式。
11. 公司所使用的开源应用程序。
12. 两家公司所拥有的相似应用。
13. 如何轻松修改应用。
14. 架构的相容性或差异性。
15. 机构之间的福利待遇差异。

除非公司是一个集团企业,并经常收购其他公司,否则尽职调查的组织工作将无比艰巨。在并购的过程中,从律师和专家那里获得专业意见是必要的。其他建议可能需要向安全和质量顾问了解,当然,可能也需要向架构专家了解信息。

到 2049 年,对于兼并、风险投资以及其他关键的商业目的,智能工具和 AI 工具的结合也可以协助做尽职调查。

3.15 认证和授权

软件人才的认证和授权是颇具争议的话题。医疗领域和法律领域的认证和授权也同样存在争议。如果认证是合理的,那么在相反的情况下,由于不当行为被取消认证资格也应该是合理的,但是这么做的话甚至会有更大的争议。

医疗认证的历史在 Paul Starr (1982 年普利策奖的得主) 的著作《The Social Transformation of American Medicine》(美国医学的社会转型)中做了阐述。目前,由于在所有的专业中,医药是最负盛名的专业,因此阅读斯塔尔的书,并考虑医疗在 19 世纪 50 年代是如何实践的,我们感同身受,就像软件当前的状况,这将是一件非常有趣的事情。

当时,医生的培训课程是两年制的,并且也没有高级专科住院实习的机会。许多医学院校以营利为目的,并没有要求学生拥有大专以上文凭,或者甚至连高中毕业文凭也没有要求。在美国,有超过一半的医生没上过大学。

大多数医生在医学院校进行培训,但是他们从来没有进入过医院,或者临床治疗实际的患者。某些神秘的医学院校除了教授“标准”的医学专题外,还教授非标准的疗法,如顺势疗法。那时也没有对医学院校进行合法认证。

医院本身既没有认证也没有受管制,同时它们也没有与医学院校进行关联。许多医院都要求,所有的患者只能由医院的医师治疗。当患者进入医院时,正规医生不能对其进行治疗,甚至不能进行观察。

Paul Starr 书中介绍了美国医学协会(American Medical Association, AMA)成立的原因,并说明了他希望改进医生的培训,并引进正规的专业、认证以及授权的原因。这需要 AMA 大约 50 年的时间来实现这些目标。

如果软件行业需要认证和授权,那么医疗认证的这种途径可能是最好的参考模式了。就像早期的医疗认证一样,在认证出现以前,对于已经进入各个领域的现有从业者来说,某种形式的“先驱”将是必需的。

我们需要考虑一个有趣的问题,那就是,软件工程有什么实际的主题是如此重要,以至于认证和授权是有价值的?在医疗领域,全科医生和内科处理大多数的患者,但在某些情况下,患者会交给专家,如肿瘤癌症、心血管病、妇产科等。目前医学专业有 24 个认证委员会以及大约 60 种专业。

对软件工程来说,也许以下一些主题足够重要,需要专门的培训和考试委员会的认证:

1. 常规的软件工程。
2. 软件维护工程。
3. 软件安全工程。
4. 软件质量工程。
5. 大型系统工程(大于 1 万个功能点)。
6. 嵌入式软件工程。
7. 商业软件工程。
8. 医学软件工程。

9. 武器系统软件工程。

10. 人工智能软件工程。

也有一些专业的主题，软件工程师可能会涉及也可能不会涉及：

1. 软件的指标和度量。

2. 软件的合同和诉讼。

3. 软件的专利和知识产权。

4. 软件的客户培训。

5. 软件的文档和帮助信息。

6. 软件的客户支持。

7. 软件的测试和静态分析。

8. 软件的配置控制。

9. 软件的可重用性。

10. 软件的病理学与法医学分析。

11. 软件的尽职调查。

12. 数据和业务规则的挖掘。

13. 智能工具的部署。

随着时间的推移，还可能会添加其他主题。目前情况下考虑的主题是：正规的训练是必要的，并且认证或授权也许是有价值的。

截至 2009 年，十几种软件主题都有形式多样的自愿性认证。其中一些包括软件项目管理、功能点计数（不同系列的功能点）、六西格玛、测试（几个不同组织的不同认证）、Zachman 架构法以及质量保证等。

截至 2009 年，在同一领域，似乎未经认证的从业者和已经认证的从业者之间没有任何法律上的区别。没有大量的经验数据表明认证可以极大地改进软件的性能，并以此来证明认证的价值。例外情况是，一些对照研究表明，认证的功能点计数人员比未经认证的功能点计数人员有更高的精度。

毫无疑问，到了 2049 年，对软件来说，将存在其他形式的认证，但是软件的培训、授权以及认证是否达到和医疗一样的水平，我们还无从知晓。

在 2009 年，由于成本超支以及进度超期，大约有 1/3 的大型软件项目被终止。大多数的项目在完成时都已延期了，并且也超出了预算。在交付时，几乎所有的软件应用都含有大量的缺陷以及许多非常严重的安全漏洞。

很明显，从目前情况看，软件工程在 2009 年还不能称为一个真正的工程学科。如果软件工程是一门真正的学科，就不会有这么多的故障、灾难、质量问题、安全漏洞以及成本超支等。

如果软件工程应该成为一个授权和认证的职业，专业过失无疑会成为一个重要问题。只有当软件人才的培训和绩效使得项目的失败率低于 1%，并且缺陷去除效率接近 99% 时，“软件工程”的表现才能足够好，最终才能降低由于专业过失所带来指控的可能性。事实上，2049 年很可能是一个乐观的日期。

3.16 软件诉讼

无论经济如何衰退,诉讼似乎并不属于经济领域,并且各种抱怨的诉讼将明显增加。在违反合同的软件纠纷诉讼中,笔者曾经作为专家证人。以下列出许多类型的诉讼,但这只是一部分:

1. 侵犯专利或版权。
2. 对软件资产价值的税务诉讼。
3. 窃取知识产权。
4. 剽窃或复制代码和文档片段。
5. 违反竞业限制协议。
6. 违反保密协议。
7. 软件供应商的欺诈和虚假陈述。
8. 软件外包商的欺诈和失实陈述。
9. 软件故障引起的损害、死亡或受伤。
10. 收回由于计算机欺诈被盗的资产。
11. 花费过多的时间来修复缺陷,违反保证条款。
12. 由于软件的不当管制,对管理人员的诉讼。
13. 由于公司对安全的松懈态度导致数据被窃取,从而引发了对公司的诉讼。
14. 对大公司的反垄断起诉,如微软。
15. 对高管的财务违规行为进行的欺诈指控和诉讼。

当涉及信息的搜索和整合时,法律和诉讼领域可以为软件界提供许多帮助。在2009年, Lexis-Nexis 法律参考公司已经能够从3000多个来源搜索到500多万份文件。不仅如此,法律信息已经能够交叉索引,并且比软件文献更容易追踪相关的主题。

在一些诉讼中,笔者曾作为专家证人参与其中。在工作中,笔者发现,看出庭律师如何着手准备是很有趣的。从总体上看,与任何软件工程师或软件经理所知道的一个新软件应用中的问题相比,一个优秀的诉讼律师会知道更多案例相关的问题。出现以上情况,部分是由于优秀的自动化工具已经可以搜索法律材料,部分是由于律师事务所中的组织和支持团队,在收集关键数据上,律师助理给执业律师提供支持。

对于结构化软件开发来说,诉讼的结构甚至可能是一个有用的模型。诉讼中的第一个文件是由原告提出的申诉。由于大多数软件应用的出现,都是因为对旧应用的不满或对特定做法的不满,因此对于初始需求来说,使用法律申诉的格式可能会是一个很好的模型。

在诉讼的调查举证期间,被告、原告经常会在专家证人的协助下,为律师准备的书面问题提供书面回答。对于收集软件项目更详细的需求以及初步的设计信息来说,举证阶段将是一个很好的模型。

在首次申诉和调查举证完成阶段之间,聘请的专家证人通常会处理具体议题,并协助律师以书面形式书写宣誓证词。专家们还要写自己的专家意见的报告,其会用到他们行业主题方面的知识。对于软件的诉讼,经常会用到质量控制和软件成本方面的专家。在软件

项目中,让外部专家提出一些重要议题(如安全性和质量)也是很有益的,因为内部人员可能完全没有资格。

发现阶段完成后,诉讼的下一阶段就涉及证言,这时,双方的律师会向被告、原告、证人以及专家提问。虽然某些方面的质量功能展开(QFD)和联合应用设计(JAD)有少量的相似之处,他们涉及许多个人观点,并且也尝试在面对面会议中解决关键问题,但是在大多数的软件开发项目中,目前还没有确切等价的证词。

在诉讼中,证词会让真正的问题水落石出。优秀的诉讼律师通过证词找到对方的案件和人员所有可能的弱点。对大型软件项目来说,有一种形式的证词可能是非常有用的,顾问会向相关干系人、软件构架师和设计师提问,这时顾问充当的是原告和被告双方的律师。

对于软件来说,这种方法的价值是,有人扮演了魔鬼代言人的角色,并在架构、开发计划、成本估算、安全计划、质量计划以及其他的主题上寻找弱点,而这些主题往往会造成重大软件项目的失败。通常情况下,软件项目都是片面的,并且往往都是由狂热者驱动的,但狂热者却对负面的事实没有任何兴趣。原告和被告双方的律师和专家证人之间的对抗可能会在软件项目失控之前,阻止很多危险的软件项目;不然的话,项目花那么多钱,最终却被取消,对公司来说将是一个重大的损失。

对于寻找类似案件的事实,如果我们在寻找类似项目的事实和见解时,可以达到和律师一样的复杂程度,那么这对软件来说将是很有用的。

一旦智能工具和专家系统在软件开发和软件维护中开始发挥作用,当然,它们在软件诉讼中也会发挥作用。以下几个例子展示了智能工具和专家系统如何支持软件诉讼:

- 对于软件信息来说, Lexis-Nexis 和其他诉讼支援团体使用的搜索引擎已经超过了等值搜索功能了。
- 对于税务案件,现在已经开始使用软件成本估算工具了,它们用来模拟应用的原始开发成本,以及应用未能收集的历史数据。
- 在诉讼中,对质量低劣或损害的指控是原告索赔的部分,但是无论是原告方还是被告方,都应该尽可能地增加代码段静态分析的严谨性。
- 一种新的诉讼可能会很快出现。这种诉讼主要针对公司的数据被盗的情况,数据被盗窃使得成千上万的客户或受害者的身份被盗用,并且也造成了其他一些方面的损失。由于现实的罪犯可能生活在其他国家(或者甚至是其他国家政府),因此抓捕就显得相当困难了。由此公司可能会遭受无端的指责,大家都会认为数据被窃取的罪魁祸首是公司的安全防范措施不足。这是继发性损失的一种形式,美国的法院几乎不允许出现这种诉讼。但是,如果应该启动这种诉讼,这种诉讼可能会迅速增加。静态分析和专家系统可以分析应用中被窃取的数据,并找出应用的安全漏洞。
- 为了获得涉案应用与相似应用的比较信息,对于遗留应用创建功能点总数来说,自动估算规模的方法能够适用于某几种类型的诉讼,如税务案例、违反合同等。应用的规模与质量和生产率息息相关,因此确定应用的规模对某几种类型的诉讼来说是很有用的。
- 一种新型的软件成本估算工具(本章所使用的例子)可以预测外包以及软件开发承包

发生诉讼的可能性。当尝试交付、安装并使用有错误的软件时，同样的工具能够预测用户会遇到的缺陷和问题。

- 在本章中所使用的软件成本估算工具，在 2009 年，已经以原型的形式开始运作了，它可以预测原告和被告的诉讼费用。无论任何一方进入诉讼，只要对所付出的努力、所涉及的费用、正常业务活动的中断以及软件项目冻结的可能性有任何想法，评估工具都会起作用的。除非他们庭外和解，不然的话，在审判期间，原型会不断评估法律的效力、专家证人的努力以及员工和高管的努力。
- 一旦涉及代码的非法复制，静态分析工具会在不同的应用程序中找到相同的代码段。（偶尔，在盗窃的情况下，软件公司会故意在应用中插入无害的错误或不寻常的代码组合，以作为告密者的触发器。这些可以使用智能工具来识别，或者作为静态分析工具的特殊因素。）
- 在专利侵权案例中，静态分析工具和其他形式智能工具的结合，可以搜索出原有的知识以及类似的设计。
- 在违反合同或案例涉及难以接受的索赔（质量水平方面）时，软件基准和软件质量基准可以用来支持专家的意见。
- 对于诉讼来说，证词陈述是面对面的，法庭速记员会记下所有的陈述。然而，对于软件会议和事实收集，2049 年的时候也许会使用更方便的方法。许多会议可以在虚拟环境中举行，参与者通过虚拟化身进行互动，这些虚拟化身可能是符号也可能是真正参与者的图像。法庭速记员当然不需要关心有关软件需求和设计的普通讨论，但它可能会对关键的讨论感兴趣，如 Dragon Naturally Speaking 技术的使用。然后使用专家系统对原始文本的讨论进行分析，以获得业务规则、关键算法、安全和质量问题以及其他相关事实等。
- 一个功能强大的分析引擎倘若可以检查源代码；执行静态分析；执行循环复杂度和基本复杂度的分析；找出可能被非法复制的代码段；量化功能点的规模；检查测试覆盖率；发现易错模块；寻找安全漏洞；寻找性能瓶颈；并执行其他类型的严格分析；那么对诉讼来说，它将是一个非常有力的支持工具，当然，对遗留应用的维护也是很有用的。这样的工具目前已经存在了，但所有的工具并不为一家公司独有，它们尚未完全融合成一个单一的工具。

发生软件诉讼是很不幸的，费用昂贵，对正常的业务来说也将是破坏性的。希望今后能在质量控制和可重用材料的使用上有所改善，以减少违反合同和授权的案例。然而，无论软件是如何构建和维护的，税务案件、专利侵权、窃取知识产权以及违反就业协议的行为可能依旧会发生。

总之，在信息是如何收集、如何分析以及如何使用上，软件行业应密切关注法律界人士。

3.17 总结

假如可以检查类似的应用，并挖掘相似应用中的业务规则和算法，那么 2009 年到 2049

年之间，软件开发将发生巨变。另一个巨变是，从个性化设计和手工编码转变成可重用设计和可重用代码段的构建。

当这些新变化发生时，我们将需要一种新的设计和开发工具，通过数据挖掘和模式匹配，可以分析现有的应用并提取有价值的信息。我们要求智能工具能够在互联网上搜索有用的数据和专利信息。除了专利，我们还要求智能工具能获得政府规章、法律、国际标准以及其他信息。

最后一个变化是，到了 2049 年，每一个应用程序能够定期地搜索并收集生产率、质量以及其他基准方面的数据。目前，有些工具已经可以达到这些目的，如 ISBSG 的问卷调查，但它们并没有得到广泛的使用。

软件行业的目标应该是，通过零缺陷的材料来自动地构建应用，来代替通过非个性化的设计以及劳动力密集的手工编码来构建应用。

由于全球经济衰退还会再持续一年，因此所有公司都需要找到降低软件开发和维护成本的办法。通过手工编码来开发软件，已经接近开发生产力的极限了，并且也很难达到质量标准和安全标准。因此，我们需要更加自动化的方法来取代定制设计和手工编码。

维护和项目组合也需要降低成本，并且，智能工具和专家系统在改善软件组合的经济和安全方面正面临关键时期，其目标是可以提取潜在的业务规则，并挖掘质量和安全漏洞。

参考文献

- Festinger, Leon. *A Theory of Cognitive Dissonance*. Palo Alto, CA: Stanford University Press, 1957.
- Kuhn, Thomas. *The Structure of Scientific Revolutions*. Chicago: University of Chicago Press, 1970.
- Pressman, Roger. *Software Engineering - A Practitioners' Approach, Sixth Edition*. New York: McGraw-Hill, 2005. (最新中文版《软件工程：实践者的研究方法》(原书第 7 版)，郑人杰，马素霞译，机械工业出版社 2011 年 5 月出版)
- Starr, Paul. *The Social Transformation of American Medicine*. Basic Books, 1982.
- Strassmann, Paul. *The Squandered Computer*. Stamford, CT: Information Economics Press, 1997.

软件人员如何学习新技能

4.1 引言

2008年金融危机加上随之而来的2009年全球经济衰退,使专业技术人员培训产生了深刻变化。诸如电子学习(e-learning)等低成本培训方法快速扩张,而高成本的培训方法,如现场技术大会(live conference)和教室形式(class-room)的面对面培训以及某些形式的纸质出版物,则急速减少。所幸的是,电子学习相关技术的有效性也正在不断提高。

软件技术的进步非常迅速,几乎每个月都会出现新的编程语言,几乎每天都会出现新的编程工具,而每年都会出现许多新的软件开发方法。

软件技术变化速度之快意味着软件专业从业人员需要不断学习新技能。那么,哪些渠道可以用于软件人员学习新技能呢?这些可用的学习方法到底有多好呢?将来可能会出现哪些新的学习方式呢?

即使新学习方法不断改善,软件教育现状仍然远远落后于使软件人员与最新专业技术状态保持同步的真正需要,这方面有许多非常重要的话题可谈。最明显的不足包括以下几个方面:

1. 构建低风险应用的软件安全实践。
2. 将交付缺陷降至最少的软件质量控制实践。
3. 有效经济分析的软件度量方法和指标。
4. 按时交付的软件估算和规划方法。
5. 优化使用可重用组件的软件架构。
6. 有效改造遗留应用的软件方法。
7. 软件技术评估和技术转移方法。
8. 软件知识产权保护。

如果软件行业要从一种艺术形式(art form)成长为一个真正的工程学科,需要快速填补软件专业人员培训和教育方面的差距与不足。

由于经济持续不景气,参加现场技术大会的人数持续下降,一些组织取消了内部培训,还解雇了一些软件培训师。这些都可能是永久性的变化。从2009年到将来不确定的某个时间,电子学习和网络研讨会(webinar)很可能会成为职业教育的主要渠道。

随着经济持续衰退,较新的教育方式(如虚拟环境、虚拟化身和文本挖掘)可能继续增长。除了以上提到成本低的特点,这些较新的方法相比其他方法更具实际优势。

4.2 软件学习渠道的演变

软件领域像人类历史上的任何行业一样,正在快速发展出各种新技术。这意味着软件专业人员需要非常快速地学习新知识、新技能。

截至2009年,在美国,编程或者软件开发与维护等技术领域大约有260万人就业,大概有28万名软件经理。在相关专业领域(比如软件销售、客户支持、软件技术文档写作以及许多其他工作)可能另有110万辅助人员就业。

如果考虑所有软件相关职位,美国共有超过380万名专业软件技术人员。欧洲的软件专业人员总数会略微超过美国。全世界软件人员总数正在接近1800万人。对于印度、中国和俄罗斯,精确的软件人员数量尚不可知,但是这三个国家的软件人员总数加起来,很可能等于美国软件人员总人数,且这三个国家的软件相关工作正在快速增长。在全球范围内,所有软件人才都需要不断更新自己的知识以跟上快速发展的软件技术。

2008年金融危机和2009年及以后的经济衰退,很可能会破坏正常的软件教育渠道。为节省资金,许多公司会削减培训经费。由于裁员或公司破产,大量软件人才失去工作。因此,现场技术大会参会者可能会大幅减少,同时商业教育课程的学习者也会减少。长期经济衰退在大学教育和研究生教育上的影响尚不确定。如果助学贷款及资金来源不中断,正常工作机会缺乏可能实际上会增加大学和研究生入学率。

很大程度上由于对于赞助者和受教育学生的低成本,网络研讨会和在线教育渠道很可能会扩大。实际上,网络研讨会爆炸式增长如此之快,因而人们需要一个按主题进行组织、包含了所有网络研讨会的集中式研讨会目录。任何一天,美国都会至少举办50场网络研讨会,而且毫无疑问,这个数目会很快上升到几百。

能够想象到的是,经济衰退会导致大量新学习手段如雨后天春笋般涌现,比如虚拟现实“教室”、文本挖掘以把纸质文档转变为可Web访问文档以及把信息从纸质形式转换为电子书和可Web访问形式。

1995年首次发布软件人员教育渠道报告时,只有10个主要渠道可用于软件从业人员获取新信息(见表4-1)。就有效性和成本而言,这些方法各不相同。

今天,软件从业人员有17种方法用于获取新信息。新形式的电子书籍、博客、Twitter、网页浏览(Web Browsing)、网络研讨会和模拟网站(如“Second Life”)已经进入新学习方法之列。除此之外,谷歌和微软正在把数以百万计的纸质文档转换成在线可Web访问的文档。

当前报告使用一种新方式来评估各种学习

表 4-1 1995 年可用的软件培训渠道

1	内部培训 (In-house education)
2	商业培训 (Commercial education)
3	供应商培训 (Vendor education)
4	大学教育 (University education)
5	活页簿自学 (Self-study from workbooks)
6	CD/DVD 自学
7	现场技术大会
8	通过互联网和万维网的网络在线教育
9	书籍
10	期刊杂志

方法。每一种学习方法都以4个因子进行排名,每个因子取值范围从1(最好)到16(最差)。它们分别是:

1. 学习成本
2. 学习效率
3. 学习有效性
4. 信息新颖性(Currency of information)

数值“1”是每个教育类别的最高排名或得分。所有可用教育方法按每个主题依次降序排列。

第1个因子是学习成本,无需过多解释。简单而言,指一个学生使用这种教育渠道可能要付出的费用。成本从几乎免费(如网页浏览)到非常昂贵(如加入一所重点大学或者进入研究生院学习)而差异显著。

第2个因子是学习效率,是指向学生传授一定数量新知识所需要的时间。数值“1”表明该方法是最有效率的学习形式。因此在线教育和网络浏览是最快速的学习方法。

第3个因子是学习有效性,指一个学习渠道可以传输给学生的信息容量。分数“1”表明该方法是最有效的学习形式。因此大学教育和内部培训的现场指导者比其他学习渠道传递更多信息。

第4个因子是信息新颖性,指正在被传递信息的平均年龄^①。分数为“1”的学习渠道,被评为最高排名或者具有最新的信息数据。对于这个因子,网络在线教育资源往往具有最新数据信息,其次是现场技术大会及商业培训。大学教育则排名靠后。

表4-2列出了2009年的评估报告中评估的17种教育渠道。

从1995年到2009年,两种计算机辅助学习方式——网络浏览和网络研讨会——不仅添加到了上述列表中,在成本、信息新颖性和效率类别上,还成功达到了最高排名,但在有效性方面它们仍处于中等位置。

随着经济衰退的加深和延续以及技术不断发生变化,我们预期会在学习方法上看到许多变化,尤其是降低成本上的变化,同时希望有效性达到更高。

表4-2 截至2009年1月软件学习渠道排名

平均分	教育形式	成本	效率	有效性	新颖性
3.00	网络浏览	1	1	9	1
3.25	网络研讨会	3	2	6	2
3.50	电子书籍	4	3	3	4
5.25	内部培训	9	4	1	7
6.00	CD/DVD 自学	4	3	7	10
7.25	供应商培训	13	6	5	5
7.25	商业培训	14	5	4	6
7.50	Wiki 站点	2	9	16	3
8.25	现场技术大会	12	8	8	5

① 这是个比喻,指信息的存在时间。——译者注

(续)

平均分	教育形式	成本	效率	有效性	新颖性
9.00	模拟网站	8	7	13	8
10.25	通过书籍自学	5	13	12	11
10.25	期刊杂志	7	11	14	9
10.75	在职培训	11	10	10	12
11.75	导师指导	10	12	11	14
12.00	书籍	6	14	15	13
12.25	本科教育	15	15	3	16
12.25	研究生教育	16	16	2	15

4.3 软件工程师当前需要学习哪些技术主题

表 4-3 是过去几年在软件社区中脱颖而出的新术语、缩略词和缩写词抽样。这些术语揭示了 2009 年软件人员需要了解的各类技术的些许端倪,某些技术即使晚至 2000 年时还不存在。

表 4-3 软件知识领域

1	敏捷开发	15	CRM (Customer Relationship Management, 客户关系管理)
2	ASP (Application Service Provider, 应用服务提供商, 通过万维网获取软件)	16	数据挖掘
3	自动静态分析	17	数据质量
4	自动化测试	18	数据仓库
5	B2B (Business to Business 的首字母缩写或网上业务)	19	Dot-Com (一家电子商务公司)
6	BPR (Business Process Reengineering, 业务流程再造)	20	电子商务
7	Caja [⊖]	21	电子学习
8	客户端-服务器计算	22	E 编程语言
9	云计算	23	EA (企业架构)
10	计算机安全	24	ERP (企业资源规划)
11	CMM (软件能力成熟度模型)	25	极限编程
12	CMMI (能力成熟度模型集成)	26	设计和代码的正式审查
13	COSMIC [⊖] (来自加拿大和欧洲的功能点变种方法)	27	功能点度量
14	配置控制	28	GUI (图形用户界面)

⊖ 谷歌公司 Ben Laurie 发起的一个项目,旨在制订一个 JavaScript 语言子集和最佳编程指导方针,约束 JavaScript 程序员,编写更加安全、更加合理的 JS 代码。详情参见 <http://code.google.com/p/google-caja/>。——译者注

⊖ COSMIC 功能点是 IFPUG 功能点的一种重要变种。COSMIC 一词代表 Common Software Measurement International Consortium,通用软件测量国际联盟。——译者注

(续)

29	黑客防御系统	53	可重用性
30	HTML(超文本标记语言)	54	Scrum
31	I-CASE(集成计算机辅助软件工程)	55	安全漏洞
32	IE(Information Engineering, 信息工程)	56	软件即服务(SaaS)
33	ISO(国际标准化组织)	57	SOA(面向服务架构)
34	ISP(因特网服务提供商)	58	SOAP(简单对象访问协议)
35	ITIL(信息技术基础架构库)	59	软件六西格玛
36	JAD(联合应用设计)	60	静态分析
37	Java	61	故事点
38	JavaScript	62	供应链集成
39	OO(面向对象)	63	TCO(总拥有成本)
40	OLAP(在线分析与处理)	64	TickIT ^①
41	OLE(对象连接与嵌入)	65	TQM(全面质量管理)
42	正交缺陷跟踪	66	TSP(团队软件过程)
43	过程改进	67	信任计算
44	PSP(个体软件过程)	68	UML(统一建模语言)
45	QFD(质量功能展开)	69	用例
46	RAD(快速应用开发)	70	用例点数
47	REST(具象状态传输)	71	虚拟化
48	RPC(远程过程调用)	72	基于Web的应用
49	Ruby	73	Web对象点数
50	Ruby with Rails	74	Wiki 站点
51	RUP(Rational 统一过程)	75	XML(可扩展标记语言)
52	遗留应用改造		

① TickIT 项目是英国标准研究所以 ISO 9000-3 标准为基础, 为软件开发系统的注册提供的一种方法。TickIT 项目帮助软件企业建立与其业务过程相关的质量体系, 并使该体系满足 ISO 9001 的要求。——译者注

纵使有 75 个条目之多, 该列表也只涵盖了截至 2009 年软件工程领域不超过全部知识 30% 的部分主题。事实上, 软件工程领域有超过 700 种已知编程语言及其派生语言, 超过 50 种软件设计方法, 以及至少十几种已命名的软件开发方法, 这里还没有提及任何上述方法组合所产生的几乎无限多的衍生方法。

表 4-3 中的主题主要关注软件工程领域。软件工程之外的许多主题也会影响到软件行业, 比如资产管理、软件许可、知识产权保护、软件项目管理、ITIL (the Information Technology Infrastructure Library, 信息技术基础架构库) 及数以百计的其他主题。

该列表中的每个主题都相对较新并且相对较为复杂。软件人员如何才能与最新软件技术保持同步? 更重要的是, 软件人员怎样才能真正学会在他们的实际工作中足够好地有效使用这些新概念?

既然不是每个新技术主题都适合所有软件项目, 并且有些技术主题还相互排斥, 那么

软件工程师和软件项目经理对这些主题有足够多的了解以便于为他们的项目选择合适方法就变得至关重要。

一个有意思的说法是，为什么传统技术主题趋向没落以致再无人使用？原因之一是软件工程师的新技术主题太多了。例如，大多数软件人员花在培训上的时间非常有限，如果他们必须在一个新技术主题（如敏捷开发）和一个传统技术主题（如设计审查）中做出选择，相对于旧主题，新技术主题非常有可能受到青睐。

16 世纪前十年英国金融家托马斯·格雷欣爵士（Sir Thomas Gresham）指出，当流通中的两种货币相对于同一个固定参照标准（如黄金）具有不同价值时，人们会囤积具有较高价值的货币而在市场上使用较低价值的货币。格雷欣定律相当简洁地描述了这种现象：“劣币驱逐良币”。

对于软件培训，学生学习时间有限，在选择是学习一门全新的技术，还是学习一门旧技术时，新技术课程往往受到青睐。软件技术教育方面的格雷欣定律体现为“新技术驱逐旧技术”。

新技术课程比旧技术课程更加受欢迎这一事实，与这两种主题的有效性 & 价值无关。这仅仅是一个现象，即新技术主题似乎比旧技术主题更受青睐。

诸如需求、设计和编码的正式审查等一些较旧的技术主题，仍然是迄今为止已开发的最有效的软件缺陷去除方式之一，而且对缺陷预防也很有价值。然而，正式审查课程却很少能吸引到与诸如测试驱动开发、敏捷开发和自动化测试等新技术课程一样多的学习者。

在本书中，评估各种教育渠道整体有效性的方法是，组合以下几个方面的数据：每个教育渠道的学生数目、学生在培训材料上的满意度以及学生真正学到新技能的能力。

本节的排名来自笔者及其公司所做的访谈和客户基准研究。总体而言，笔者访问了大约 600 家公司，其中包括大约 150 家大到足以列入《财富》500 强的公司。此外，还访问了大概 35 个包括州级和国家级的政府机构。在走访的公司中，超过十数家企业雇用了至少 1 万名软件人员。例如，包括 IBM、微软和 EDS（Electronic Data Systems，电子数据系统公司）等主要软件雇主。这些公司中，超过 100 家公司雇用的软件人员超过 1000 人。

4.4 软件工程专家

大约在 2009 年，笔者曾受委托对大型企业中就职的软件专业人员进行研究。参与这项研究的企业和组织中就包括 AT&T（美国电话电报公司）、美国空军、德州仪器公司和 IBM。笔者和他的同事们对这些企业进行了深入走访与详细问卷调查。研究者还电话联系了另一些公司以获取研究所需信息，或者在他们对其他企业的评估和基准咨询研究中获取研究所需信息。目前，针对软件技术人员的研究仍在继续，最新获得的研究结果数据已公布在笔者出版的书籍《Software Assessments, Benchmarks, and Best Practices》(Addison-Wesley Professional, 2000) 中。

最初的研究不仅发现了有关软件行业劳动力的一些有趣事情，还发现了很多令人迷惑的现象。例如：

- 大企业雇用了种类繁多的软件专家 (specialist)。
- 小企业则较少雇用专家而较多使用知识面较宽、技术全面的通才 (generalist)。
- 系统软件和信息技术 (IT) 组织使用不同类型的专家。
- 通用头衔, 比如“技术人员”, 使这项研究的人口数据充满歧义。
- 工作职位所使用的头衔因公司而异。
- 相同头衔的工作职位所完成的工作内容, 也因公司而异。
- 电子商务领域正在产生大量新的各类工作职位、头衔。
- 人力资源部门很少有精确的软件雇员数据。
- 软件行业除了拥有“计算机科学”和“软件工程”之类学术学位的人员之外, 还有许多拥有其他学科学术学位的从业者。
- 有些软件从业人员往往不把自己算作软件专业人员之列。

最后一点相当令人惊奇! 很多编写嵌入式软件的人, 所接受的教育是电子工程师或者机械工程师。即使他们的主要工作实际上是开发软件, 这些嵌入式工程师中的一些人仍拒绝称呼自己为“程序员”或“软件工程师”。

在受访的一家公司, 数以百计的工程师在为自动化专用系统开发嵌入式软件, 但是他们所拥有的学术学位是其他工程学科的, 所以他们没有被人力资源部门认为是软件工作者。当这些工程师接受采访时, 他们宁愿以他们所拥有的学术工程学位而不愿以他们实际所从事的工作来称呼自己。与电子工程师、电信工程师、自动化工程师等职位头衔相比, 软件工程师的头衔好像地位较低一些。

在对其进行问卷调查的公司和政府组织中, 没有哪个组织的人力资源部门能够精确地列举出他们所雇用软件从业人员的数量或者他们工作职位的头衔名称。在走访的几家公司和政府机构中, 他们的人力资源部门根本没有那些现在已经成为软件社区全部职位一部分的专家职位的工作描述, 或者甚至没有一些主流工作职位 (如“软件工程师”或“程序员”) 的说明。

假如你有兴趣去了解做某项具体工作 (比如“业务分析师”) 所必须要求的职责和知识技能, 你去访问那些似乎有可能聘请这种专家的 10 家公司, 截至 2009 年, 你可能会获得如下所述结果。

- 雇用业务分析师的公司中的两家, 使用由公司人力资源组织维护的相似工作职位描述。
- 雇用业务分析师的公司中的三家, 使用本公司独特的、由本公司维护或者公司人力资源部门也不清楚的工作职位描述。
- 三家公司使用“业务分析师”职位头衔, 但是本部门、公司或者公司人力资源部门没有关于该职位的任何书面说明。
- 两家公司有人似乎在做业务分析师的工作, 但是他们使用不同的工作职位名称, 比如“技术人员”或者“咨询分析师”。而这两个职位名称具有被许多其他专家使用的通用工作描述。

鉴于今天这种局面的不确定性,要想确定诸如“有多少软件雇员在为大型组织工作”这样的基本事实都非常困难,而要弄清楚诸如“所雇用专家的确切种类和数量”这些更精确的数据,截至 2009 年,几乎是完全不可能的。要确定这些专家所需要的特定类型知识和培训并非完全不可能,但是这些需求有成千上万种局部变化,并且没有统一的、一致的整体模式可循。

定义具体职位并招聘软件技术人员的职责经常被委托给相关软件部门高管和经理,人力资源组织则成为次要角色并经常只涉及处理录取确认书(offer)和将新员工信息录入到不同的工资及行政系统中去这样的工作。

该项研究所进行的访问显示,在典型的大型高科技公司,仅仅使用由人力资源组织提供的雇员数据所做的软件行业统计分析会低估软件行业从业人员真实数量达 30%。这是因为,像嵌入式软件开发、质量保证、技术文档作家和测试专家等专业职位并没有包含在“程序员”或“软件工程师”的统计数目内。

另一个令人迷惑的地方是通用职位头衔的使用,比如可以归入超过 20 多个专业的“技术人员”,这些专业包括软件工程师、电子工程师、电信工程师、航空电子工程师以及许多其他技术职位,比如业务分析师和质量保证人员。在一些公司,“技术人员”头衔包括了硬件工程师、软件工程师和技术支持人员,比如技术文档作家。

上述不足和人力资源组织真实数据的缺失解释了某些杂志发布的软件人口研究结果与政府组织(如劳工统计局)发布的数据之间的矛盾。毫无疑问,就人力资源组织提供的报告资料而言,许多研究还算是精确,但是这些研究无法处理所用原始数据的错误。

4.5 软件专业的种类

软件专家的种类总数超过了 100 种且还在增长。目前观察到的软件专业整体分类可分为如下 5 个独立领域。

- 特定软件工具、方法或者语言(如 Java 或面向对象)领域的专家。
- 特定商业、行业或科技领域(如银行)的专家。
- 软件开发支撑支持类任务(如测试、质量保证或者文档工作)领域的专家。
- 软件开发管理类任务(如规划、评估和度量)领域的专家。
- 软件生命周期各个组成部分(如开发和维护)的专家。

表 4-4 列出了目前大型企业和政府机构的软件人口与评定研究过程中观察到的主要软件专家种类。尽管表 4-4 包含了 115 种职位或形式的专家(仍然不是全部),但在研究的公司中,还没有任何一家雇用了超过 50 种可以正确识别的软件专家。那些使用“技术人员”作为通用职位头衔的公司,可能还有在笔者的研究中没有识别出来的其他专家。

另外一个最近在十几家专门从事软件业务的公司里发现的新职位是“福音布道师”或“传道师”。这是一个非常耐人寻味的头衔,它表明,软件技术选择往往是一个以信仰为基础的,而不是以科学或知识为基础。

表 4-4 大型软件组织中的软件专家

1	会计/金融专家	42	图形用户界面 (GUI) 专家
2	敏捷开发专家	43	黑客专家 (防御目的)
3	软件架构师 (软件/系统)	44	人为因素专家
4	软件评估专家	45	信息工程专家
5	审计专家	46	讲师 (管理主题)
6	布德里奇美国国家质量奖 (Baldrige Award) [⊖]	47	讲师 (软件主题)
	获奖专家	48	讲师 (质量主题)
7	软件基线专家	49	讲师 (安全主题)
8	软件基准专家	50	软件集成专家
9	业务分析师	51	互联网专家
10	业务流程再造 (BPR) 专家	52	ISO 认证与标准专家
11	CMMI 专家	53	JAD 专家
12	软件复杂性专家	54	知识专家
13	软件组件开发专家	55	库专家 (项目库)
14	配置控制专家	56	诉讼支持专家
15	软件成本估算专家	57	软件维护专家
16	咨询专家 (各种主题)	58	市场专家
17	课程规划专家	59	技术人员 (多个专业)
18	客户联络专家	60	软件度量 (Measurement) 专家
19	客户支持专家	61	软件度量指标 (Metric) 专家
20	数据库管理专家	62	微代码专家
21	数据中心支持专家	63	多媒体专家
22	数据质量专家	64	本地化 (Nationalization) 专家
23	数据仓库专家	65	网络维护专家
24	决策支持专家	66	网络专家 (局域网)
25	软件开发专家	67	网络专家 (广域网)
26	分布式系统专家	68	网络专家 (无线网)
27	领域专家	69	神经网络专家
28	教育专家 (各种主题)	70	面向对象专家
29	嵌入式系统专家	71	外包评价专家
30	加密专家	72	软件包评估 (Package evaluation) 专家
31	ERP 专家	73	软件性能专家
32	软件框架专家	74	个体软件过程 (PSP) 专家
33	专家系统专家	75	项目成本分析专家
34	功能点专家 (COSMIC 认证)	76	软件项目经理
35	功能点专家 (IFPUG 认证)	77	项目规划专家
36	功能点专家 (芬兰认证)	78	过程改进专家
37	功能点专家 (荷兰认证)	79	软件生产力专家
38	通才 (Generalists, 执行软件相关的各种各样任务的人)	80	软件质量保证 (QA) 专家
39	国际化和本地化专家	81	质量功能展开 (QFD) 专家
40	图形艺术专家	82	软件质量度量专家
41	图形制作专家		

⊖ 布德里奇美国国家质量奖由美国国会于 1987 年设立, 并以美国前商务部部长 Malcolm Baldrige 的名字命名。这一奖项依据 7 个类别的绩效标准对企业进行评判, 旨在促进企业提高质量意识。其绩效标准分类模型, 常用来评估企业管理系统和识别主要改进领域。——译者注

(续)

83	快速应用开发 (RAD) 专家	100	软件标准化专家 (ANSI、IEEE 等)
84	Rational 统一过程 (RUP) 专家	101	统计专家
85	高级研究员	102	系统分析专家
86	软件可靠性专家	103	系统支持专家
87	存储库专家	104	技术评估专家
88	再工程 (Reengineering) 专家	105	团队软件过程 (TSP) 专家
89	更新改造专家	106	技术翻译专家
90	逆向工程专家	107	技术文档编写专家
91	软件可重用性专家	108	软件测试专家
92	风险管理专家	109	测试库控制专家
93	销售专家	110	全面质量管理 (TQM) 专家
94	销售支持专家	111	软件价值分析专家
95	软件范围经理	112	虚拟现实专家
96	Scrum Master	113	Web 开发专家
97	软件安全专家	114	Web 页面设计专家
98	面向服务架构 (SOA) 专家	115	Web 网站管理员
99	软件六西格玛专家		

有“布道师”头衔的环境里通常会出现相当新的技术概念，例如“Java 布道师”或者“Linux 布道师”。这个头衔非常常见，所以几年前，一名微软员工发起建立了“全球技术布道师网络”(GNoTE)^①。

布道师的头衔似乎是褒义的，它经常与有趣的新观念相联系。就新想法在广泛推广和采用之前如何检验与测试而言，它确实凸显了软件工程、医药工程和传统旧工程领域的某些不同之处。在软件方面，相比于基于实验或者实际使用产生的经验数据，有魅力的领导者似乎更加广泛地参与到技术选择和技术转变之中。

Guy Kawasaki (前苹果员工) 在他 1991 年出版的书籍《Selling the Dream》中可能首次公开在软件领域使用“布道师”一词。但是在访谈中，Kawasaki 说，当他首次加入苹果公司时苹果已经在使用“布道师”这个头衔了，所以并不是他发起使用这个头衔的。若如此，苹果公司则可能是这个有趣头衔的源头。

其他一些已经出现的软件职位头衔的起源也值得关注。六西格玛社区已经采用了一些来自武术的术语，并使用“黄带”、“绿带”和“黑带”来表示熟练使用六西格玛的不同等级。

如果许可授权和圆桌认证会发生在未来的软件工程中，那么推测国家执业资格考试如何处理诸如“布道师”或者“黑带”等这样的头衔将很有意思。

专业化和公司规模大小之间有着有趣的关联，同样，专业化和行业之间也有某种有意思的关联。

通常，拥有总数超过 1000 名软件工作者的大型公司有最大数目的各种专家。就职称而言，少于 25 名软件人员的小公司则可能根本就没有专家，即使可能在软件人员之间有一些

① GNoTE 的全文是 Global Network of Technology Evangelists, 发音是 gee-Note。它是由来自微软、Sun 和雅虎的技术布道师共同发起建立的。其目的是为了满足提高工业和学术领域对技术布道师的认识，建立技术布道师社区，彼此分享技术布道师的工作经验。详细信息请参见 GNoTE 官方网站。——译者注

类似工作在做。

高科技行业，如电信、国防和航空业，会比服务类行业和零售、批发、财务及保险业的公司雇用更多专家。

就技术而言，软件领域正在迅速扩大。在软件行业里，一个人无法通晓有效地完成某项工作所需全部信息的时代已经来临。当这种情形发生时，进行专业化分工就是再正常不过的应对之策了。

将表 4-3 中的 75 种技术主题和表 4-4 中的 115 种专家进行组合，其结果将是一个包含 8625 种结果的庞大矩阵。因为每一个结果都有它自己独一无二的知识和信息要求，由此可见，要想跟上现代软件工程各个主题的发展速度是件非常困难的事情。

如果 1000 个软件工程主题和 250 种职业的软件及业务专家进行组合，其结果矩阵可能会有 250 000 个结果之多。很明显，未加辅助的人类智慧无法囊括以上矩阵所有结果的全部知识。人们需要一些简化形式和一种正式知识分支的分类方法以便有序地保持这种大规模信息。毫无疑问，这里也需要智能化工具和专家系统为特殊活动与职业提取并分析这些知识信息。

专业化已经出现在最具学术性的学科里，比如医药和法律。实际上，对科学来说专业化早已是普遍现象。化学、物理、生物学和地质学等专业学科的出现都少于 200 年的历史。在这些专业化领域取得发展之前，它们是被那些具有通用头衔的人（如“自然哲学家”）带着学术和科学的愿望识别出来的。

如果软件专业许可和认证成为现实，那么存在超过 115 种不同专家是不很现实的。这几乎是 2009 年已存在医学类专业数目的 3 倍，比法律专家数目的 4 倍还多。

削减和整合某些专业似乎非常必要，其目的是保持软件行业专家总数与法律和医学行业专家种类的数目相一致。也就是说，将保留少于 50 种正式软件专业领域和大概 25 个有着显著数量从业人员的专业。

有趣的是，当下的软件教育课程似乎更多地着眼于通才而非专业人才。例如，访谈到的绝大多数质量保证人员通过在职学习和内部课程来学习测试与质量控制的内容，而不是在大学里学习这些知识。

当前，万维网的爆炸式增长及成为大公司主要焦点的电子商务的出现，正日益扩大着工作职位头衔的数目。例如，“网站管理员”和“网页开发员”头衔的出现尚不超过 10 年时间。

在软件领域，新专业化形式的出现非常快速：几乎每年都会出现超过 5 种新专业化领域。第 3 章讨论了软件认证和专业化主题，提供了必需专家最小可能数目的建议。这是一个识别软件职业和相应知识及学习渠道之间相互关系的尝试，它要求以专业水准来进行这种尝试。

4.6 专家与普通软件人员的大概比率

专业化领域的一个主要话题是，需要多少种专家来支持软件通才社区的全部工作？

表 4-5 使用术语“通才”来指代那些从事软件开发编程工作的雇员和那些工作职位头衔是“程序员”、“程序员 / 分析员”或者“软件工程师”的雇员。

表 4-5 专家和普通软件人员的大概比率

专家职业	专家 / 通才比值	通才百分比	专家职业	专家 / 通才比值	通才百分比
软件维护和功能增强专家	1 : 4	25.0%	网络专家 (局域、广域)	1 : 50	2.0%
软件测试专家	1 : 8	12.5%	软件性能专家	1 : 75	1.3%
软件系统分析师	1 : 12	12.0%	软件架构专家	1 : 75	1.3%
技术文档编写专家	1 : 15	6.6%	软件成本估算专家	1 : 100	1.0%
软件业务分析员	1 : 20	5.0%	软件可重用性专家	1 : 100	1.0%
软件质量保证专家	1 : 25	4.0%	软件范围经理	1 : 125	0.8%
数据库管理专家	1 : 25	4.0%	软件包采购专家	1 : 150	0.6%
配置管理专家	1 : 30	3.3%	软件安全专家	1 : 175	0.6%
系统软件支持专家	1 : 30	3.3%	软件过程改进专家	1 : 200	0.5%
功能点计算专家	1 : 50	2.0%	教育和培训专家	1 : 250	0.4%
软件集成专家	1 : 50	2.0%	软件标准化专家	1 : 300	0.3%
软件度量专家	1 : 50	2.0%			

尽管也经常负责其他工作, 比如需求分析、设计、测试以及可能还要创建用户文档, “通才”这一分类是指那些主要负责为计算机编程的软件工作者。

专业化主题的深入讨论才刚刚开始, 因而表 4-5 中的比率可能有较大误差。事实上, 对于某些种类的专业化, 还没有规范的比率数据可用。并非表 4-5 中的所有专家分类都会出现在同一家公司。表中数据以降序排列, 来自雇用了总数超过 1000 名软件人员的大型企业。

尽管笔者研究软件专业化的目的是探讨大公司中的软件职业, 但其结果之一是为人力资源组织所保存的数据推荐一些重要的本质性的改进建议。

通常由人力资源组织所维护的人口统计数据并不足以预测软件职业长期发展趋势, 甚至不足以推测当前的就业情况。这些数据似乎不完整, 因而许多政府机构和大型信息技术公司 (如 Gartner 集团) 所公布的软件职业人口统计研究结果效果甚微。如果我们甚至都不知道今天有多少软件人员就业, 这将很难以现在的软件职业就业形势去讨论未来的需要。

不但软件职业人口数据有巨大不足, 而且在所有这些专业化职位真正所需要知道的、用于以高水平职业能力来完成他们工作的知识和技能上, 也存在着巨大差距。这些不足和差距影响着大学本科教育和研究生教育课程, 也影响着每个单一的学习渠道。

据笔者推测, 软件行业还不是一个真正的工程行业, 因而它有大量的非正式专业。软件正在慢慢从一门手艺 (craft) 或者一种艺术形式 (art form) 不断进化, 因此无论是所需知识种类, 还是需要以职业的行为方式使用这些知识的专家类型, 都远未达到稳定状态。

4.7 评估软件工程师所使用的学习渠道

近 50 年来, 主要软件雇主一直在为其员工提供持续不断的培训和教育以试图使其员工

跟得上技术的发展。典型地，一流软件雇主具有以下几个的共同特征。

1. 为新员工提供 4 ~ 10 周的密集培训。
2. 每年 5 ~ 10 天的年度内部培训。
3. 每年 1 ~ 3 个外部商业研讨会。
4. 月度或者至少季度的新技术主题网络研讨会。
5. 越来越多用于自学的 DVD 培训课程库。
6. 藏书丰富的技术书籍和期刊杂志图书馆。
7. 管理人员和技术人员的学费偿还计划。
8. 基于 Web 的技术站点访问权。
9. 订阅软件和管理杂志，如《CrossTalk》。
10. 订阅行政管理杂志，如《CIO》。
11. 订阅信息提供商（如 Gartner 集团）发布的信息。
12. 越来越多地使用网络研讨会、电子书籍和基于 Web 的信息系统以共享信息。

遗憾的是，随着很多公司苦苦挣扎以努力维持正常运营，2009 年的经济衰退很可能导致很多教育方法出现重大萎缩。不仅很多教育方法可能会被削减，很多培训讲师也许面临着下岗的命运。自由职业或创业者数目可能降低到相当低的程度以至于培训需求也会降低。

因为有多种教育渠道可供选择，所以思考每种可供选择的教育渠道的主题、优点及缺点是很有意思的事情。下面的教育渠道以它们整体评分来排名，“1”为最好。下面将深入地讨论表 4-2 中列出的 17 个教育渠道。

第 1 名：网络浏览

成本 = 1；效率 = 1；有效性 = 9；新颖性 = 1；整体评分 = 3.00

预测：快速扩张。

网络浏览（又称“网上冲浪”）使用诸如 Google 或者 Ask 等搜索引擎，是目前世界上能搞清楚几乎所有技术主题最快速且经济有效的方法。通过使用几个搜索短语，如“软件质量”或者“软件成本估计”，数分钟之内，成百万页面的信息就唾手可得。

网络浏览的缺点是获得的信息过于杂乱无章、混乱无序以及好坏各种信息鱼龙混杂。即便如此，不仅对软件工程师，对任何知识工作者，网络浏览都是现代最强有力的学习工具。

很多网络门户网站提供了大量连接到其他相关信息来源的链接。门户网站之一是信息技术指数与生产力学会（ITMPI）网站，它提供了很多领域的广泛主题，比如软件工程、质量保证、项目管理和遗留应用维护等。详细信息请参见 <http://www.ITMPI.org>。

在各种学术类链接站点中，最完整的一个是英国格拉摩根（Glamorgan）大学戴夫·W·席恩（Dave W. Farthing）的网站（<http://www.comp.glam.ac.uk>）。这个有趣的网站包含大量项目管理类站点的链接和导航到许多项目管理书籍出版者站点的链接。

第三个有用的软件主题门户网站是软件工程研究所（Software Engineering Institute, SEI）的网站，某种程度上认为该组织是个政府或者国防部门。然而，该网站涵盖了大量吸引人的主题，并提供了许多有用的链接。详细信息请参见 <http://www.SEI.org>。

在软件领域,网络浏览在按主题信息整合、交叉索引和知识提取方面仍然比较原始,有待更大程度的提高。

第2名:网络研讨会、播客(Podcast)和电子学习

成本=3;效率=2;有效性=6;新颖性=2;整体评分=3.25

预测:使用范围快速扩展;有效性得到提高。

长久以来,使用计算机进行技术培训早已不是稀罕事,而新技术也正在对这种培训方式迅速进行改进。数年之内,学生们就将在虚拟环境中广泛参加各种培训,而这种虚拟环境也增添了不少新学习方法。

网络研讨会是一种新形式的研讨会,它正迅速流行起来。在网络研讨会中,演讲者和参会听众都分散在他们各自的办公室或家里,所有人都使用他们自己的电脑来参加研讨会。网络研讨会的主办方连线所有参与者并提供后台技术支持以防任何人掉线。主办方还会收集对网络研讨会的评价以不断改进。

播客类似于网络研讨会,不同之处在于播客可能会也可能不会现场直播。它们会记录下来,这些信息可以在任何时候按需索取。播客还可以包括小测验和考试、自动阅卷并根据学生们的答案自动导航到学习材料的不同部分。

在网络研讨会上,主要演讲人会同会议协调者用电话与参加者进行通信,但PowerPoint幻灯片或其他计算机生成信息将会在主演讲人或者会议协调者的控制下出现在与会者屏幕上。

由于不需要出差,网络研讨会的效率和成本都相当不错。其新颖性评分也相当好。截至2008年,有效性仍处于中等水准,但会快速增长。当前,网络研讨会主要用于持续时间在90分钟以内、单一目的的讲座或讲课。

任何一天,网络研讨会或播客的数量都在迅速扩大,各种不同组织会同时提供50个这样的网络研讨会。人们几乎不可能弄清楚网络研讨会的确切数目。网络研讨会带给行业的价值是一个非营利的信息分类目录,它允许任何公司或大学至少提前两个月预先公告他们所有网络研讨会的时间表和会议内容摘要。

将来可以预期的是,由于网络研讨会和播客的成本效益极佳,全部持续时间在10~20小时的课程都会迁移到网络研讨会方式上来。某些时候,使用虚拟化身的虚拟环境也将参与进来,而后,电子学习将很有可能成为人类历史上比任何其他培训方式更为有效的培训方式。

目前,网络研讨会技术仍然不够成熟。断线、容量小和断断续续的电话问题非常常见。例如,VOIP电话呼叫经常根本就无法工作。但毫无疑问,这些问题将会在未来一到两年内得到解决。

理论上可以设想,全球范围内全部课程在学生和讲师根本就不见面的情况下只使用网络研讨会与计算机通信进行授课,这是有可能的。

无论是抽象的虚拟化身还是使用实际真人头像,从今天的网络研讨会到学生和老师都可以看见彼此的虚拟教室,实际上只有一步之遥。

一些大学(如麻省理工学院(MIT))已经将视频和音频连接集成到了演讲厅,这样,远程学员可以参与到课堂现场讨论中来。只需要很小的一步,就可以将这种技术扩展到手持设备,或者把演讲和现场讨论记录下来以备后用。

除了只参加网络研讨会然后说拜拜外,精明的公司也注意到同数百名潜在客户的联系也是一个很好的市场机会。参加者可以向讲师登记或者公司可以主办此项活动以获得更多信息。

实际上,也可以邀请网络研讨会参加者参与其他形式的信息推广,比如 Wiki 站点,还有博客、Twitter。这些都是发表个人看法和意见的重要渠道。

在线学习的长期发展方向是令人鼓舞的。可用信息越来越多,而且越来越多的人将能够彼此沟通和分享想法,而在现实生活中不再需要出差或者面对面接触。

最后,虚拟化身、虚拟现实和使用智能化工具提取与整合知识改进方法的某些组合,将使人们有能力通过在线和网络访问方式提供从高中到研究生教育的全部课程。毫无疑问,无线技术也会应用到教育中来,信息也将在手持设备如智能手机上可用。

第3名:电子书

成本=4;效率=3;有效性=3;新颖性=4;整体评分=3.50

预测:使用范围快速扩展;有效性得到提高;费用降低。

电子书游走在教育和娱乐行业边缘已经有20年的历史了。过去,与纸质图书相比,甚至与屏幕更加容易阅读的普通计算机相比,电子书籍的典型特征是屏幕昏暗、界面别扭、下载速度慢、内容有限和使用不便。

早就有机构如 Project Gutenberg 努力使纸质书籍可以以 HTML (Hypertext Markup Language, 超文本标记语言)、PDA (Personal Digital Assistant, 个人数字助理)、Word 或者其他网络可访问的方式为人们所使用。随着谷歌、微软以及电子书市场的其他主要公司努力尝试使用自动文本转换工具把几乎100%的纸版图书转换成网络可访问格式,这种趋势现在正在激增。不用说,这种尝试已经在版权、知识产权以及对书籍作者及版权持有者的补偿方面产生了严重纠纷。

目前已经有巨量在线文本内容可以获取,同时也存在一些技术问题需要解决或改进,比如交叉引用、摘要、索引、包含内容丰富的分类目录等。但是,这种网络可访问文本的趋势正快速增长,并将持续增长。

有趣的是,法律界人士似乎已经在很多方面稍微走在了软件行业前面,这些方面包括知识的集中整理、交叉引用以及让真正需要信息的专业人士获得信息。例如, Lexis-Nexis 公司^①的数据库已经连接到了大约 30 000 个数据来源的超过 500 万个文件^②。目前还没有哪个组织具有如此组织良好的软件信息汇集。

① 律商联讯公司,世界著名的数据库提供商,全球许多著名法学院、律师事务所、高科技公司的法务部门都在使用该公司的数据库。目前该数据库连接至 40 亿个文件、11 439 个数据库以及 36 000 个来源,资料每日更新。其内容包括法律研究内容、新闻报纸、杂志、学术期刊以及企业界信息。——译者注。

② 2009 年的数据。——译者注

最近,从2007年和2008年开始,亚马逊和索尼公司已经改变了电子书结构而采用了一种新模式,这种新模式可以解决旧版电子书存在的很多问题,同时还引入了一些很好很强大的新功能。

不仅在软件学习上,而且在所有其他科学学科的学习上,新的亚马逊 Kindle 和索尼 PR-505 都具有非常大的机会获得成功。其最好的特性包括优秀的屏幕清晰度、非常快的下载速度(包括无线连接)、更长的电池寿命及显著改善的用户界面。此外,这些新设备还具有某些甚至比纸质书籍更优越的特性,包括永久保存注释的能力、从一本书到另一本书跳过相同主题内容的能力以及当出版商那里有新材料可用时频繁更新的能力。

使用电子书籍作为软件学习工具,读者可以很容易地选择和下载软件项目管理的前10大好书、软件质量的前10大好书、软件维护的前10大好书、软件成本估算的前10大好书、软件开发的前10大好书,并且可以同时拥有这些书籍。

对于学院和大学,可以很容易地为每学期的每门课程下载所有相关书籍,并且可以同时使用所有这些书籍。

当前索尼 PR-505 和亚马逊 Kindle 的模式已经相当不错,但正如所有新产品一样,可以预期未来几年这些产品将迅速出现改进。可以预见,最多5年内,电子书籍将会令传统纸版图书的市场份额大幅削减。还可以预期的某些电子书特性包括处理图像和下载动画的能力、从主要技术杂志订阅电子书籍以及内置相关网站的链接。

第4名:内部培训

成本=9;效率=4;有效性=1;新颖性=7;整体评分=5.25

预测:仍然有效,但因经济衰退而趋于减少。

因为由全球经济危机及2008、2009年经济衰退而导致的裁员和部分大公司成本削减,内部培训很可能会趋于减少,更不用说一些大企业申请破产及倒闭了。

从我们调查问卷开始的1985年到金融危机和经济衰退之前的2007年,内部培训教育一直在整体有效性方面保持第一。现在,内部培训教育仍然是有效性排名第1和效率排名第4的培训方式。这意味着,通过内部培训,大量信息可以很容易、快速地传递给学员们。

然而,这种教育渠道只在如IBM、微软、谷歌以及类似的大公司才有。笔者估计,当前在足够大的公司和组织中工作的大约一半美国软件人才可以接受这种内部培训,也就是说,超过160万美国软件专业人士可以获得这种渠道的培训。

2001~2004年的研究表明,这种渠道的教育培训数量在下降。由于部分讲师下岗或提前退休,2008~2009年的经济衰退导致一些内部培训被取消。

除此之外,入门级职位招聘减少也相应减少了对刚加入大公司初级职位的应届大学毕业生进行培训的需求。随着经济衰退延长和加深,2010年,内部培训很可能继续减少。如果其他教育渠道(如虚拟教育和网络研讨会)继续扩张,内部培训的辉煌时期可能已经过去。

一些大的软件雇主,如埃森哲、AT&T(美国电话电报公司)、EDS(电子数据系统公司)、IBM、微软和一些其他公司拥有针对软件从业人员和经理们的内部培训课程,这些课程比大多数美国大学的课程更加多元化和时髦。一位ITT(International Telephone and Telegraph

Corporation, 国际电话电报公司)的前董事会主席观察到,美国《财富》500强企业所拥有的软件讲师比所有美国大学讲师总数之和还要多。这些大公司中的雇员们享有的内部培训时间,比所有其他培训渠道之和还要多。

大公司的内部培训课程通常都非常集中和密集。一个普通课程会持续2~3个工作日,每天从早上8:30开始到下午4:30结束。然而,有时候为了优化安排学生的学习时间,有些课程会一直持续到晚上。

从对某些课程及其参加者的观察可知,大公司的内部培训是学习现有技术的方式方法中最有效的方法。

内部培训的另一个好处是,很容易得到参加相关课程培训的批准。获得参加公司内部培训批准需要的书面工作要远少于那些具有学费退款计划的大学本科课程。

即便在2009年,笔者仍然不确定内部培训是否会恢复到它在20世纪80年代和90年代所具有的重要性。经济前景太不明朗,无法对内部培训这种教育渠道进行长期预测。

关于内部培训的一个有趣发现是它对软件开发生产力的影响。每年为软件工程师提供10天培训课程的公司,其年度生产率要高于那些没有任何培训课程的公司,即便是为培训课程预留10个工作日。这些培训的价值以更高的生产率得以体现。

虽然以上发现的详细信息还没有深入探讨过,且可能没有办法证明这种关系,但有意思的是,那些每年提供10天培训课程的公司比那些没有任何培训的公司具有较低的人员自愿离职率。很显然,公司培训对员工的工作满意度会产生有益影响。

第5名:使用CD-ROM或者DVD自学

成本=4;效率=3;有效性=7;新颖性=10;整体评分=6.00

预测:使用中缓慢改进;有效性方面改进较快。

自学课程形式行将被CD-ROM或者DVD方法所改变,但也可能被电子书籍改变。旧形式自学课程由活页笔记本组装成的教程材料、练习和小测验组成。新的CD-ROM或DVD自学课程包括以前方法的所有组成部分,以及超文本链接和巨量的补充信息。

实际上,最新的DVD培训形式允许在课程中添加新内容。这种方法是新兴的,几乎没有任何经验数据可查。当它广泛普及时,其新颖性评分应该会快速攀升。

因为可以包括图像、动画、语音和其他主题,所以通过DVD完全交互式学习是个令人兴奋的前景。然而,制作成本和把各种信息放在一起制作成一个有效DVD课程所需要的技巧要显著大于传统的工作簿。这种成本对于学生来说并不是特别高,但制作DVD的成本就比较高。

直到大约1995年,潜在学生手中的CD-ROM驱动器数量仍低于需要的最低水平。而且仅有的CD-ROM驱动器中,很多还是老旧的单速驱动器,访问速度缓慢、颠簸不稳。

然而到2009年,笔者估计超过99%的软件专业人士办公电脑上都有DVD或CD-ROM驱动器。(超轻上网本设备重量不超过3磅,往往缺少DVD或CD-ROM驱动器。同样,一些涉及高度保密工作的组织禁止任何形式的可移动媒介,因而也可能没有DVD或者CD-ROM驱动器。)

截至 2009 年年初,笔者估计有超过 25 万名软件专业人士通过 CD-ROM 或者 DVD 参加过各类自学课程。由于经济困难和生产成本问题,目前只有大概不到 125 个这类 DVD 或 CD-ROM 自学课程可用。

随着 21 世纪前进的脚步,使用 CD-ROM 和 DVD 的自学课程有可能在数量上得到提高及有效性上得以改善。用户可以在飞机上携带和使用的笔记本电脑带有轻巧便携的 DVD 或 CD-ROM 阅读器,这将给 CD-ROM 或 DVD 这种自学渠道带来一个极大提升。这也同样适用于电子书籍。

较新技术不仅允许 DVD 内容下载到计算机上,还可以下载到电视机、iPod、智能电话和其他手持设备上。

新出现的蓝光格式 DVD 可能会改善 DVD 教育的互动能力,但是目前由于蓝光产品生产成本低相当高,这种想法仍然远未实现。

第 6 名:商业培训

成本=14;效率=5;有效性=4;新颖性=6;整体评分=7.25

预测:由于经济衰退而呈现下降。

2008 年经济危机和 2009 年及以后持续不断的经济衰退,对商业培训产生了令人沮丧的影响。学生数量不断减少,而航空旅行也变得越来越昂贵。毫无疑问,更低成本的学习方法(如电子学习)将开始取代正常的商业教育。

多年来,商业培训整体效益始终排名第二。软件领域是商业培训的重要子行业之一,到 2009 年,其仍排名第 5 且相当有效。

这个子行业里面的公司包括 Computer Aid (CAI)、Construx、Coverity、Cutter、Data-Tech、Digital Consulting Inc.(DCI)、Delphi、FasTrak、the Quality Assurance Institute(QAI)、Learning Tree、Technology Transfer Institute (TTI) 以及很多其他在全美甚至全球范围教授课程的公司。这些公司既提供技术类课程,也提供管理类课程。

以高水准运作的是专业化信息技术公司,如 Gartner 集团和 Forrester 研究公司。这些公司既提供标准的研究报告,也为客户提供定制化的专题研究。这些高端公司的主要客户群是行业内公司的决策者和高级管理层。与这些目标市场相一致的是这些研究的费用和成本高得惊人,因此它们所能吸引到的客户主要是大型企业和大型政府机构的软件高管,比如州或者主要城市政府部门的 CIO。

非营利性组织也提供一些收费培训。例如,非营利性的国际功能点用户组(International Function Point Users Group, IFPUG)就提供功能点相关主题的培训和研讨班。项目管理协会(Project Management Institute, PMI)也提供商业教育,软件工程研究所(Software Engineering Institute, SEI)也是同样。国际软件基准组织(International Software Benchmarking Standards Group, ISBSG)同样提供评估和基准研究相关主题的培训,并且还出版这些主题的图书和专题论文。

数以百计的本地公司和成千上万的个人咨询顾问在很多公司基于合同讲授公司内部课程,有时候也提供一些公开课。许多课程是由非营利性组织负责提供的,这些非营利组织

有 ACM (Association for Computing Machinery, 美国计算机协会)、DPMA (Data Processing Management Association, 数据处理管理协会)、IEEE (Institute of Electrical and Electronics Engineers, 国际电气与电子工程师协会)、IFPUG、SEI 以及许多其他组织。

据笔者估计, 大约有 50 万美国软件专业人士每人每年参加至少一个商业软件研讨班。然而, 从 2009 年起, 因为全球经济衰退, 这种情况将会出现大幅下滑。

由于主要商业培训机构是以商业方式来运作其培训课程的, 因此他们必须相当优秀以维持生存。较大培训教育机构的一个主要卖点就是使用名人做关键课程的讲师。例如, 一些知名的行业大牛, 如 Bill Curtis、Chris Date、Tom DeMarco、Tim Lister、Howard Rubin、Ed Yourdon、Watts Humphrey、Gerry Weinberg 博士、James Martin 博士以及 Carma McClure 博士, 他们都通过商业培训公司对外提供课程和研讨会。

典型的商业研讨会持续两天, 单人费用 895 美元, 共 50 个学生。商业培训微小但比较流行的特点是选用非常好的物理教学设施。很多商业软件课程都开设在阿斯彭、奥兰多、凤凰城或者旧金山等地的度假酒店里。

但是, 必须考虑 2009 年及以后的经济衰退。近几年来, 涉及去其他城市旅行的商务活动一定程度上减少了且从未完全恢复过来。随着经济衰退的影响变得越来越广泛, 商务活动还会进一步减少。

但无论如何, 商业培训的主要优势依然存在, 主要有以下两方面:

- 比较时髦、新颖的主题课程是最畅销的。
- 一流讲师的课程是最畅销的。

这意味着商业研讨会很可能涵盖从大学甚至公司内部培训课程都无法获得的材料, 也意味着学生有机会与软件领域的一些思想领袖进行互动。例如, 2009 年, 商业培训渠道比大多数其他渠道更有可能覆盖到当下热点的技术主题, 如敏捷开发、软件六西格玛和基于 Web 的主题。

正如 SPR 的评估和基准研究中提到的, 商业培训已经在处于行业软件生产力和质量水平前四分之一的公司里广泛使用。换句话说, 那些其管理者和技术工作者都期望最佳业绩的公司认识到, 他们需要最优秀的教育家。

第 7 名: 供应商培训

成本 = 13; 效率 = 6; 有效性 = 5; 新颖性 = 5; 整体评分 = 7.25

预测: 由于经济衰退而呈现下降趋势。

以前软件供应商培训的整体有效性排在第 3 名, 2009 年则排到了第 6 名。这并不是因为部分软件厂商的培训质量出现下滑, 而是由于网络研讨会和 DVD 技术的快速出现和发展。

近 50 年来软件供应商提供的培训一直是软件世界主要的教育形式。由于供应商培训在电子表格和文字处理等工具方面被非软件专业人士采用, 因此学生总数非常庞大。然而, 在软件领域, 笔者估计大约有 50 万软件专业人士每年参加至少一门供应商课程。

过去, 当硬件和软件还捆绑在一起销售时, 供应商培训通常是免费的。今天, 当作为市场营销计划一部分时, 一些供应商的培训仍然免费。通常, 当客户购买了软件工具或者

软件包而这些工具或软件包又需要专门培训才能掌握使用方法时，供应商培训就会同时出售给这些客户。

几乎每款大型商业软件都结构复杂、难以使用。即便是基础应用软件，如文字处理软件或者电子表格软件，现在也包含非常多不易掌握的功能。这样，诸如 Artemis、IBM、Oracle 和 CA^①等大型软件公司都提供收费培训作为他们所提供的一部分。

由于软件规模大小、功能集以及软件产品的复杂性等因素，每个主要软件厂商现在都为其用户提供某些种类的培训。对于非常流行的常用工具，例如 Microsoft Word、WordPerfect、Excel、Artemis Views、KnowledgePlan 等，可能有相当数量的当地咨询机构甚至高中和大学课程补充了供应商提供的培训。

对于非常大型的软件，比如来自 Oracle 和 SAP 的 ERP 软件包，如果没有要么来自软件供应商要么来自支持这些软件包的专业教育机构的广泛培训，要想学会并使用这些应用几乎是不可能的。

供应商培训是复杂技术主题教育的中流砥柱，这些复杂主题包括学习如何部署和使用来自 BAAN、Oracle、PeopleSoft、SAP 以及其他供应商的 ERP 软件包。

供应商提供的培训良莠不齐，但总的来说，这些培训在推动客户前进和运行相应应用上起到了值得称赞的作用。

软件供应商培训通常比商业培训便宜很多，而且有效成本有时每人每天少于 100 美元。供应商培训经常在公司自有软件的前提下提供，通常都很便利。但另一方面，你也无法期望这些供应商培训的全体教员都是由大牌讲师组成。

但是，诸如电子学习等新方法甚至比供应商培训还要便宜，且有利于学生在一天 24 小时任何方便的时间参加课程学习。因而，未来几年供应商培训将呈下降趋势，电子学习方法将成为主要教育力量。

更进一步，随着经济衰退的深入和延续，通常处于培训一线的讲师可能由于成本削减而失业、下岗。现场讲师指导形式的供应商培训正在进入一个衰落期。

由于软件包继续增加许多新功能且更加复杂，一直到下个世纪，供应商提供的培训仍将是软件世界一个稳定良好的教育特色，但是它可能会从现场指导转向电子学习或是使用虚拟化身的虚拟环境。

2008 ~ 2009 年的经济低迷也对供应商培训产生了不利影响。一些供应商厂商由于销售减少亏损，所提供的课程自然也减少了。其他的一些软件厂商被人收购，而另一些则歇了业。尽管课程和讲师减少，供应商提供的教育仍然是软件教学的重要渠道，即使供应商、学生和课程的数目都在下降。

第 8 名：现场技术大会

成本 = 12；效率 = 8；有效性 = 8；新颖性 = 5；整体评分 = 8.25

预测：由于经济衰退而呈下降趋势。

① 前 Computer Associates，现 CA Technologies 公司。——译者注

很遗憾,2008年金融危机和2009年及以后的经济衰退导致现场技术大会严重减少,而且这种趋势可能会继续下去。据笔者估计,由于裁员、成本削减和其他金融问题,2009年现场技术大会的与会者人数可能比2006年相同会议减少约30%。

在传授新技术的有效性方面,软件相关技术大会排名第8。然而,在突出强调吸引人的新技术方面,现场技术大会应该排名第1。遗憾的是,“9·11”灾难和自2000年以来的经济衰退叠加影响,使得很多公司削减了商务旅行预算,这也影响了现场技术大会的与会者人数。

笔者估计,2003年软件技术大会的参会者可能比1999年常态情况下少15%。然而,2006年和2007年出席技术大会的人数有所增加。本书写作时,2008年经济衰退仍然没有对许多技术大会产生影响,但毫无疑问,基于美国和世界经济状况不佳,到2008年年底前及可能在2009年,技术大会的参会者将会减少。

即便如此,每月仍然有许多重要软件技术会议举行,甚至有些月份可能有多个会议。这些技术大会可能由非营利组织发起,如美国空军 STSC (Software Technology Support Center, 软件技术支持中心)、IEEE、IFPUG、GUIDE (Global Uniform Interoperable Data Exchange, 全球统一互操作数据交换)、SHARE (Software-Hardware Asset Reuse Enterprise, 企业软硬件资产重用) 或者 ASQC (American Society for Quality Control, 美国质量控制协会),也可由商业会议公司发起,如 Computer Aid (CAI)、Cutter、数字咨询公司 (DCI)、软件生产力集团 (SPG) 或者技术转移研究所 (TTI)。也有些针对公司高管的高端会议由研究机构如 Gartner 集团或者学术界机构如哈佛商学院或斯隆管理学院发起。

此外,有些是供应商为用户举办的大会,这样的公司有苹果、微软、CA、Oracle、CADRE、COGNOS、SAS、SAP 以及类似公司。这些经常是年度活动的会议有时能够吸引到好几千名与会者。

笔者估计,2007年大约25万美国软件专业人士每人参加至少一个技术大会,而很多专业人士可能会参加多个技术大会。2008年因美国经济下滑参会人数可能会出现下降。

软件技术大会是最前沿软件技术发布和展示的地方,有时候可能还会发现某个技术是第一次出现。许多技术大会还会在主要的大会活动之前或之后举办相关培训、专题讨论,因此也会与商业培训及供应商培训有所重叠。

然而,大多数与会者去参加技术大会是去了解在其选定的领域内哪些是最新的和最激动人心的技术的。技术大会上从精彩绝伦到无聊之极的各种演讲者和技术主题都有。技术大会通常会安排很多并发举行的主题讲座(session),这样参会者可以很容易地离开一个无聊、不感兴趣的讲座而去寻找一个精彩的、有趣的讲座并听讲。

大多数会议举行面向全体与会者的主题演讲和并行举行的分主题讲座,这些分主题讲座涵盖更专业的技术主题。一个典型的美国软件大会通常举办3天,吸引200~3000名不等的参加者,并有20到超过75个主题演讲者。

除了主要演讲者和研讨会领导者,许多大会也有“供应商展示”环节,这些相关商务领域的公司可以展示他们的产品,可能会销售这些产品,或者至少获取潜在客户的联系方式。供应商为参加这样的大会而支付的费用用于支付会议行政支出,有时甚至允许大会作

为一个盈利机会而运作。

整体上,技术大会在向听众展示最先进技术方面做得很好。你很少会从一个介绍深入、熟练使用一项新技术的大会逃开,且你经常会对什么样的技术值得仔细分析有一个深刻理解。

最近几年,一些技术大会变得如此庞大以至于其论文集开始用 CD-ROM 来发布,而不再是传统笔记本或者装订成册的打印文本。

将来,现场技术大会有可能与网络研讨会或者 DVD 产品进行融合。现场活动和在线网络研讨会同时举行在技术上是可行的,这样现场观众和远程观众都可以同时看到发言者。也有可能记录大会论文集或会议录以备随后离线使用或者下载到计算机和手持设备上。

正如在互联网泡沫崩溃和早期经济衰退时所发生的一样,如果经济继续下滑且现场技术大会与会者继续流失,网络研讨会将可能成为现场技术大会的有效替代而用于现场指导。

第 9 名: Wiki 站点

成本=2;效率=9;有效性=16;新颖性=3;整体评分=7.50

预测:使用中快速增长;有效性增长。

单词“wiki”在夏威夷语中的意思是“快速”。这个术语已经开始应用在那些允许多人共同目标而参与合作的 Web 站点。

Wiki 站点是第一次出现在教育渠道列表中。这是一个新兴技术,它允许多参与者为一个共同主题或共同事业而合作。

Wiki 站点最著名的例子就是 Wikipedia (维基百科),它已成为世界上最大的百科全书。Wikipedia 中的每个词条或文章都是由志愿者编写的。读者或者其他志愿者可以修改该条目,并输入自己的思想和观点。

乍看起来,Wiki 似乎容易导致混乱且可能会有充满攻击性的言论。但是许多 Wiki 站点,包括 Wikipedia,都语调温和、内容丰富,这可能是因为读者可以立即删掉具有攻击性的评论。

一些 Wiki 站点试图在发布之前筛选和监控用户输入及修改;其他站点则简单地允许其他读者自己更正其材料。以上两种方法似乎皆行之有效。

Wiki 站点最优秀的地方是允许具有共同兴趣爱好的人们快速产生包含了他们共同积累和智慧分享的文档或者 Web 站点。这样,Wiki 站点非常适合了解技术问题,例如,敏捷开发、软件质量、软件安全、测试、维护和众多从业者需要分享新数据及经验的其他技术主题。

第 10 名: 模拟网站

成本=8;效率=7;有效性=13;新颖性=8;整体评分=9.00

预测:在使用方面快速增长;有效性快速增长。

模拟器用于教授手工操作技能,如怎样开一架飞机或如何组装一挺机枪,它已经被商业和军事组织使用多年了。

但是近几年,出现了新形式的、应用范围更广泛的模拟网站,比如 Second Life,这是一种虚拟世界,虚拟化身(计算机用户的符号替代)通过一个虚拟场景漫步于这个虚拟世界

里，并与其他虚拟化身互动及交换诸如文本文档和图像等资源。

除了培训和教育外，这种新形式的虚拟现实技术还有其他用途，但其在教育方向的真正应用才刚刚开始。例如，在虚拟空间里有可能使用虚拟化身教授软件正式设计和代码审查。不仅可以通过模拟来讲授正式审查方法，而且确实可以通过模拟来执行软件正式审查活动。

许多其他新技术也可以通过同样方式授课。当前，教程材料的生产成本相当高且过程复杂，但这两个问题应该很快会得以解决。

模拟真实世界的虚拟现实技术在未来5年内很可能成为一个快速发展的教育方法。理论上，我们可以创建一个虚拟“大学”。在这个虚拟大学里，学生和教师都可以只在网络环境下进行教学互动。

截至2009年，虽然模拟和虚拟现实技术仍然不是主流教育方法，但其获得成功的胜算仍在，它们将会在10年内上升到教育方式排名的前列。

第11名：软件期刊杂志

成本=7；效率=11；有效性=14；新颖性=9；整体评分=10.25

预测：由于经济衰退，数量会减少；有效性保持稳定；快速从纸质向在线出版格式迁移；某些杂志会迁移到电子书格式。

现在，期刊杂志可以用于亚马逊 Kindle 和其他电子书平台。我们预计，由于经济原因，电子期刊将开始取代纸质期刊。电子出版物要比纸质出版物更快捷、更便宜，而且更加环境友好。更进一步讲，电子期刊可以在几分钟内发布，所以没有纸版期刊邮寄那样的延迟。

传统软件期刊在传递技能上通常排名第14位。软件期刊的主要长处在于普及概念和展示某项技术的整体面貌，而不是该技术的深度细节。

目前软件行业有许多与软件相关的期刊杂志。有些是为获取利润且依赖广告收入而出版的商业杂志。其他许多期刊则由非营利性专业协会制作。例如，与其他 IEEE 期刊一样，《IEEE Computer》就由非营利的协会制作。

美国空军软件技术支持中心（STSC）出版了一个博学而包含广泛的软件杂志——《CrossTalk》。该杂志已经成为为数不多为争取文章深度而努力的软件杂志之一，而不是主要简短引用行业数据组成的泛泛而谈。

另一个有趣的杂志是《Cutter Journal》，其最初由软件大师 Ed Yourdon（爱德华·尤登）以“American Programmer”名字创立。与《CrossTalk》类似，《Cutter Journal》杂志发表完整的技术文章。

在众多软件期刊杂志中，很多杂志相当专业且覆盖了相当窄的领域。一个例子是国际功能点用户组（IFPUG）的杂志《Metric Views》。

许多软件期刊软件专业人士可以免费获取，这提供了潜在订阅用户，只需要填写一个调查问卷（和负责获取相应的工具或软件）即可。另一方面，有些杂志年度订阅费用可能高达1000美元。

软件专业人士常常都会订阅很多软件杂志，但实际上只会阅读其中的少数。SPR（软件

生产力研究所)估计软件技术人员每月平均订阅或访问大概4本杂志。

相对而言,很少有软件杂志包含具有持久技术价值的文章。当杂志做技术讨论时,很少会有技术深度方面的考虑。这可以理解,因为既没有哪个记者也没有哪个专业作者会在写文章之前大费周折地花费大量时间来收集、整理信息。

就技术传播而言,最好的文章经常是那些主题明确、具体、针对特定问题的文章。例如,一些杂志已有如质量保证、度量和指标、软件维护、面向对象方法、变更管理及此类主题中具体问题的讨论文章。

用处最少的文章是由记者制作的那些典型的一刀切式调查结果,这些文章大部分包含20~50个不同人的简单调查结果。这种方法很少真正能把相应话题带入公众视野。

一个有趣的新趋势是既出版在线杂志又出版纸质期刊。在这些新在线杂志中,最优秀、最有意思的一个是《Information Technology Metrics and Productivity Institute (ITMPI) Journal》。该杂志的网站是<http://www.itmpi.org>。该杂志由Computer Aid公司发布,包含如Watts Humphrey和Ed Yourdon等软件知名人士的访谈、技术文章以及数百个相关网站的引用。

随着在线交流成为信息交换的普遍媒介,软件杂志和其他以纸质印刷品为主的信息媒体都在创建与纸质印刷品并行的在线版本。

第12名:使用图书、电子书和培训材料自学

成本=5;效率=13;有效性=12;新颖性=11;整体评分=10.25

预测:由于经济衰退,使用数量会减少;有效性将保持稳定;快速从纸质文档迁移到DVD或在线信息。

在整体有效性方面,图书自学市场排名第11。传统自学课程市场并没有快速增长,但是近50年来,它始终保持相当的稳定性。笔者估计每年总共大约5万名软件专业人士参加各种自学课程。

自学材料的通常格式是活页笔记本。这种形式的自学材料对那些能够自我激励的人有效,但它往往索然无味,时常令人厌烦。有些自学课程也包括录影带或录音带。

自学课程内容的最常见主题是那些市场潜力相对较大的技术主题。这意味着诸如“COBOL编程介绍”之类的基础技术主题最有可能是自学形式的。

实际上,因为需要时间去制作自学材料以及市场的不确定性,新技术很少是自学形式的。当然总有例外,一些全新的主题,比如ISO 9000-9004标准,因其显而易见的巨大市场,已经出现了自学形式的课程。

理论上,手持设备上的电子书似乎将成为任何种类学习形式的有用媒介。然而,在笔者的客户中,截至2009年,还没有为这种手持设备生产出正式的课程材料。我们尚未见到足够多使用这种设备的学生,因而无法对这种现象发表看法。整体上,尽管它们显示出了巨大的未来潜力,手持阅读设备的使用仍然处于其非常早期的发展阶段。

在2006~2009年,可用在PDA和苹果公司iPod等手持设备上的图书和视频材料已经有了不小增长。一些教育材料已经可以在手持设备上获取,但是由于数量太少,笔者难

以做出是否有效的成熟判断。

2008年亚马逊和索尼的新款手持阅读设备比之前几个版本更加成熟。如果这些设备取得成功,这将成为电子阅读设备即将进入主流的象征。手持设备可以容纳很多书籍的事实将坚定地旅行者甚至销售人员证明其优势所在。

第13名:在职培训

成本=11;效率=10;有效性=10;新颖性=12;整体评分=10.75

预测:由于经济衰退而使用数量下降。

在职培训经常教授公司内部开发与使用的特殊技艺,也教授那些大学里很少会教的基本技巧,比如软件设计和代码的正式审查。

通过在职培训,在选定软件方法上新员工接受那些方法经验丰富使用者的指导。最有效的在职培训是那些“执行一项任务就是学习该任务的最好方法”类的主题培训。比如正式审查、质量功能展开(QFD)、学习如何使用私有功能及学习如何使用某些编程语言,这些编程语言是某些个别公司内部培训唯一的最佳之选。

在职培训的缺点是做培训的专家需要花费他们自己的工作时间(来做准备及进行培训)。这就是为什么在职培训成本相当高。同样,老员工教授给新员工的东西可能在某种程度上是过时的,这种事情时有发生。

随着经济衰退的延续和深化,在职培训将可能因裁员和机构精简而减少。剩下的人员可能不再有时间参与到大量在职培训中。

第14名:导师指导

成本=10;效率=12;有效性=11;新颖性=15;整体评分=11.75

预测:由于经济衰退而使用数量下降。

“导师指导”的概念包含新员工与老员工甚至经验更加丰富的资深员工之间的一对一关系。导师指导最常用于专用于专方法,比如,教新员工学习本地企业的标准,或者教新员工如何使用仅在一个地方使用的经过定制的工具。

导师指导经常在社会化层面上有效,但是如果要求导师从自己正常工作中拿出太多时间的话,那将花费不菲。

非正式导师指导方式很可能一直有用,特别是对以下情况尤其有效:教授新员工学习本地方法、工具、开发方法、维护方法和其他定制主题。

遗憾的是,目前还没有关于导师和被指导者具体数量的统计信息,所以这种学习渠道的使用情况和整体有效性仍然不甚清楚。

第15名:专业书籍、专著和技术报告

成本=6;效率=14;有效性=16;新颖性=13;整体评分=12.00

预测:纸质出版物减少;快速迁移到电子书、在线和网络出版物。

在美国,传播软件技能的整体有效性方面软件书籍排名第15位。较低排名并不是因为书籍本身,而是因为美国软件产业似乎并不特别有文化修养,考虑到软件工作的性质,这多少有点让人吃惊。

很多软件书籍都很优秀。例如,Barry Boehm博士的经典著作《Software Engineering Economics》(Prentice Hall)、Watts Humphrey的书籍《Team Software Processes (TSP)》(Addison Wesley)以及Roger Pressman博士的经典作品《Software Engineering - A Practitioner's Approach》(McGraw-Hill),这些书籍都有众多版本且都销售了近百万册。

即使不用作教学目的,软件书籍还主要作参考之用。一个典型的软件工程师会拥有20~100册藏书,这些藏书涵盖许多重要主题,比如,日常工作中用到的编程语言、操作系统、应用程序以及硬件等。

截至2009年,美国有300万名软件从业者,且该数字仍然还在以相当快的速度增长。不过,软件图书出版商认为售出超过1万册才算是相当好的销量。而对于软件主题图书,2.5万册的销量已经是非常难得,只有少数软件主题图书可以达到100万册的销量。

高销量软件图书经常瞄准的是最终用户市场而非专业人员。例如,Visual Basic或者Windows入门类图书的销量可以超过100万册,因为这些产品拥有大概1000万名用户,而这些用户都不是专业的软件从业人员。

从书本上学习软件技术是有可能的,但这种方式并没有像研讨会或者某种正式培训那样广泛使用。可能的例外是学习个人主题,比如Watts Humphrey的个体软件过程(Personal Software Process, PSP)方法。截至2009年,Watts的书仍是学习这个主题的首选方式。

通过书籍自学新技能的另一种情形是新编程语言领域。过去30年间,新编程语言已经以超过每月一门的速度出现,软件生产力研究所的“编程语言及其水平”表里列出了700多种编程语言。其中能成为主流编程语言的并不多,但是对于那些成为主流的编程语言,比如,Ruby语言、N语言或者E语言,有经验的程序员从书本中自学这些新语言的速度要比大多数其他方法快。

市场上很多出版商出版了许多优秀软件图书,这些出版商包括Addison Wesley、Longman、Auerbach、Dorset House、IEEE Computer Society Press、McGraw-Hill、Prentice Hall、Que、Microsoft Press等。

图书分类中还包括各种公司和行业协会出版的专著。例如,很多公司和协会都出版从50页到超过100页不等的软件相关专著,这些公司和协会有Accenture、IBM、IFPUG、ISBSG、各种IEEE协会、McKinsey、Gartner集团、Meta集团、软件工程研究所(SEI)以及软件生产力研究所(SPR)。

这些不公开发行的专著主要分发给相关的客户。根据这些专著是否作为一种市场营销手段或者是否包含客户感兴趣的专有或特殊数据,其价格从免费到超过2.5万美元不等。

市场上有许多软件专业书店,诸如Borders等通用书店内部都有大型的软件图书部门。当然,软件书籍也是大多数网络书店(如亚马逊和Barnes & Noble)的主要商品。目前可能有超过2500种技术类和250种管理类优秀图书。

虽然有大量技术图书,但是根据笔者访谈评估的反馈,许多软件经理和相当多的软件

技术人员每年所读技术书籍不超过1本或2本。

据笔者估计,软件专业人员平均每年购买4本书籍(似乎多于实际阅读的图书数量)。无论怎样,由于软件技术变化如此迅速,单靠阅读书籍来保持技术上与时俱进是很困难的。

软件及其项目管理主题的图书领域有个奇特的空白。在那些比较成熟的专业领域,比如医药和法律,有大量书籍以音频形式存在,比如磁带或者CD,这样读者可以在家里或者汽车里收听这些书籍的内容。但我们还没有碰到任何软件及其项目管理领域的音频书籍,尽管可能已经存在了。

越来越多的技术书籍已经可以在线阅读或下载,这种读书方式正在不断普及,但目前还没有达到质变的程度。据笔者估计,以在线形式获取的软件相关图书可能少于100种,而已经利用这种渠道去获取相应图书的软件专业人员可能少于12.5万人。但是在未来10年里,无论是在线图书数量还是在线图书访问量都应该会快速增长。

新的电子设备,比如亚马逊和索尼的手持书籍阅读器、苹果公司的iPhone以及各种PDA设备,可以用来存储图书,尽管这还不是图书出版商的主要市场。

当然,如果公司或政府机构维护一个藏书丰富的图书馆,那么个人拥有图书可能就变得不再必要。笔者在过去几年的评估中发现一个有趣的结果,那些拥有大型技术图书馆的公司,其年度生产率要高于那些没有这种图书馆的相同大小的公司。

拥有图书馆可能不是使这些公司具有高生产率的决定因素,但是这也能够说明,拥有技术图书馆的公司同样也往往具有高于平均水平的薪水和福利待遇,所以他们能够吸引到软件行业最优秀的人才。

大量专著和技术报告通常从20页到75页不等。通常,这些文档专注于单一技术主题,比如面向服务架构(SOA)或者市场问题。

有些公司,如美国Gartner集团或者爱尔兰Research and Markets公司,为市场制作各种专著、技术报告的纸质和在线版本,其业务量大得惊人。

这种较为简短文档的主要市场是公司图书馆,或者对商业趋势感兴趣且在公司中级别足够高而有权花费资金来订阅这些专著或技术报告的高管们,或者个人研究者。很多专著和技术报告比普通图书贵很多。

向高管们指出新兴商业趋势,技术报告这种形式相当有效。但作为学习某个特定技术主题(如测试和审查)的工具,技术报告则没有普通书籍那么有效。

纸版书籍的一个技术难题是,软件行业的变化要比技术图书修订和再版速度快得多。

随着图书出版商向电子书籍迁移,可能的新商业模式是,在电子书籍首次下载的同时,向读者出售这些书籍的更新订阅。比如,一个可下载的软件电子书籍可能售价25美元,而该书的更新订阅则售价每年10美元。这样不仅电子书籍比纸版图书更便宜,提供更新订阅也将为图书出版商提供源源不断的收入流。

第16名:大学本科教育

成本=15;效率=15;有效性=3;新颖性=16;整体评分=12.25

预测:由于经济衰退,数量可能下降;有效性保持平稳;所设课程落后于技术进步5年

以上。

根据笔者研究,大学本科教育在软件专业人员学习渠道的整体有效性方面仅排在第15名。大学课堂所教内容往往不是最新技术。通常情况下,大学课程滞后于软件科学实际发展状态达5~10年。这种滞后是由大学开发教材和课程的方式导致的。

软件领域有许多重要主题,在这些主题上,大学的软件教育水平远远落后于软件专业地位的要求。大学课程最明显的不足包括:

1. 保护高风险应用的软件安全实践。
2. 将交付缺陷降至最少的软件质量控制实践。
3. 有效经济分析的软件度量和指标。
4. 及时交付和成本控制的软件估算和规划。
5. 优化使用可重用组件的软件架构。
6. 有效改造遗留应用的软件方法。
7. 软件技术评价与技术转移。

如果软件行业要从一种艺术形式(art form)成长为一门真正的工程学科,必须尽快填补这些空白和不足。

但也有例外,大学教育并不总是滞后。许多大学已经与当地商业团体建立了相当紧密的联系,努力为学生们提供与该地区软件雇主需求相匹配的课程。例如,新泽西州的史蒂文斯理工学院已经同AT&T建立了紧密联系,正在提供包括AT&T所建议软件技术主题的硕士课程。波士顿地区的本特利学院、华盛顿大学圣路易斯分校、亚特兰大的佐治亚州立大学、圣保罗地区的圣托马斯大学以及其他毗邻大型软件制造商的学校都已经采取了类似政策,提供基于主要软件雇主需求的课程。

本科教育的一个重要优势是其所教授的内容往往是学生整个职业生涯中都用得着的。因而大学教育在学习渠道有效性方面或所传递信息的容量方面排名第3。

据笔者估计,每年大约有9.5万名美国软件专业人士会参加大学课程学习。

根据对持续软件教育进行的咨询研究,笔者还注意到一些实际问题。公司资助学费退款计划的方式往往异常烦琐,有时需要好几层管理部门的批准。而所学习的课程本身也必须与工作紧密相关。有些公司还保留了“让学生因公司需要而退出学费退款计划”的权利。

同时,学费退款政策往往以达到合格为前提^①。这并不是一项不合理的政策,但它确实提高了学生显著自付费用结束该项计划的概率。

整体而言,相比于内部培训、商业培训和供应商培训,对于实践中的软件专业人士,大学教育似乎更昂贵但也更低效。

国际电话电报公司前任总裁曾经在一次演讲中指出,一个刚毕业的软件工程师在可以被委任以重要项目职责之前,需要平均3年时间的内部培训和在职经验。这比电气或机械工程师所需要的培训时间长大约两年。最后的结论是,在学科教学的实用价值上,软件工程和计算机科学的大学课程严重滞后于传统工程学科的大学课程。

① 即只有学生成绩合格或顺利毕业才可以拿到相应的学费报销,否则学生要自己支付学费。——译者注

事实上,对几所大学软件工程和计算机科学课程所进行的快速审查也发现了学术教育中的一些严重不足。有些主题似乎从来没有教授过,这些主题包括:软件成本估算、设计和代码审查、统计质量控制、遗留应用维护、软件度量和指标、六西格玛方法、风险与价值分析、功能点以及需求分析的联合应用设计(JAD)。虽然大学教育在基础技术主题上的水平相当不错,但是项目管理相关主题就与顶尖水平相去甚远。

2008年以后,大学教育也受到了经济衰退的影响。就软件工程和计算机科学本科生和研究生的数量而言,大学教育的未来尚不确定。鉴于本书写作的时间是2009年,经济衰退是否将会提升或降低大学教育课程仍无法判断。

第17名:研究生教育

成本=16;效率=16;有效性=2;新颖性=15;整体评分=12.25

预测:由于经济衰退,数量可能减少;有效性保持稳定;所设课程滞后于技术发展5年以上。

遗憾的是,软件工程和计算机科学的研究生教育在学习渠道中常常殿后,排第16名。在有效性方面,培养研究生的研究生院确实排名第2,且它确实给研究生们传授了大量信息。

研究生教育的负面影响是,鉴于大学课程常常落后于商业和技术世界发展实践达5~10年,其传递给研究生们的大量信息可能也是陈旧、过时的。

研究生教育可以依靠更加专注于某些专题而得以提高,如软件的经济性。软件成本如此严重受制于缺陷去除费用、灾难性故障以及巨大的成本超支、严重的进度延误,因而需要教给所有MBA学生、专家级软件工程师和计算机科学研究生们最先进的软件缺陷预防、缺陷去除和软件质量经济成本的知识。

同时,软件安全问题不仅在2009年泛滥成灾,而且还正以迅猛的速度变得愈加严重。很明显,大学教育在这方面已经严重落后于实践需要,软件质量控制领域也是一样。其他关键主题如可重用材料的构建,大学本科和研究生教育同样滞后于技术发展实际。

很明显,我们需要提高大学课堂吸收、合并新材料和新技术的速度。对于快速发展的行业,如计算科学和软件行业,技术的发展变化要远远快于大学课程的变化。

对于这个问题,经济衰退的发生已经证明,大学的经济和金融课程不仅严重过时,而且还极其错误。

4.8 需要额外教育的软件领域

根据在许多因违反合同或涉及低劣质量而导致的软件诉讼中做专家证人的经验,笔者发现一些重要的、需要额外培训或再培训的软件技术主题。按照其在软件行业的重要性大数排序,表4-6展示了软件世界表现似乎有不足之处的25个主要技术领域。

表 4-6 软件培训上表现不足的技术领域

1	软件安全漏洞预防	14	软件应用的价值分析
2	遭攻击后安全恢复	15	技术分析与技术转移
3	软件质量控制(缺陷预防)	16	遗留应用改造
4	软件质量控制(缺陷去除)	17	软件需求收集与分析
5	软件质量评估	18	软件项目管理
6	软件质量度量	19	软件正式审查
7	软件度量与指标	20	软件性能分析
8	软件变更管理	21	软件应用客户支持
9	软件项目跟踪	22	软件测试用例设计
10	软件知识产权保护	23	合同与外包协议
11	软件成本、质量和进度估算	24	用户培训
12	软件构件重用	25	用户文档
13	风险分析与风险缓解		

与这些主题相关的软件故障和缺陷在软件世界普遍存在,大型软件尤其如此。从大型软件项目频频失败甚至司空见惯的项目成本超支、进度延误现象可以得出结论,软件相关的技术主题培训迫切需要进行重大改进和提高。

4.9 软件学习的新动向

全球经济衰退导致成千上万的企业为了维持经营而被迫削减成本。几乎所有的劳动力密集型活动(如培训)都需要严格仔细审查。

同时,虚拟现实技术、电子书籍以及通过手持设备的信息分发技术都在复杂性和用户数量方面有所增加。

也许10年内,经济衰退的压力和技术变革的结合将会在软件学习方法上产生严重分化。基于Web的在线信息、电子书籍以及手持设备毫无疑问地将会取代大量纸质阅读材料。

此外,虚拟现实技术可能会引入虚拟教室和模拟大学,学生和老师在这里通过自己的虚拟化身进行交互而无需在真实环境中面对面沟通。

智能工具和专家系统变得越来越复杂,这可能会提高搜寻大量网上信息的能力。谷歌和微软等公司正在快速将纸质书籍与文档转换为在线文本格式的事实也说明信息访问的方式正在改变。

然而,就虚拟信息的组织和访问而言,在达到与法律及医药行业同等的易用性和复杂性之前,软件行业还有很长的路要走。例如,截至2009年,还没有相当于Lexis-Nexis法律咨询公司这样的软件公司和组织。

将来几年,软件学习方法的改变将经历与印刷媒体和电视行业所带来的同样深刻的变化。但是,与较为成熟领域(如医学和法律)的信息质量相比,软件信息的质量仍然不佳。生产力、进度、质量和成本的量化数据严重匮乏,这使得软件开发活动更像是一门手艺而非一个真正的工程行业。

然而,信息挖掘技术、知识整合技术以及使知识更加易于获取的技术正快速改进。人们对软件学习方法和信息传递的整体预测是积极的,但毫无疑问,将来展示信息的形式将与过去极为不同。

4.10 总结

无论任何技术领域,要想始终跟上技术的最新发展都是很困难的。随着 21 世纪前进的脚步,软件技术正在快速发展,软件人员要时刻保持与时俱进将更加困难。

软件社区有 17 种不同的学习渠道可用于获取新的软件技术信息。历史上最有效的学习渠道是大公司的内部培训,但这种渠道只提供给该公司的雇员。网络研讨会和基于 Web 的学习方式在复杂性方面迅速发展,目前已经非常便宜。

虚拟现实技术和模拟网站的使用是另一个令人激动的前景。在模拟的虚拟教室中虚拟学生化身参与课堂互动学习在技术上完全可行。

在其他渠道中,两个渠道未来似乎有很大潜力:使用 CD-ROM 或 DVD 技术自学和使用万维网或信息工具的在线学习。这些学习渠道中的信息内容和用户数量正迅速扩大。正如在亚马逊和索尼的新一代电子书阅读器上已经证实了的那样,使用无线连接和手持设备的学习渠道可能也会加入到学习方法行列。

随着互联网和在线服务使用量的增长,作为大规模国际交流渠道的副产品,未来可能会出现全新的教育方法。将来,通过卫星广播可能会提供某些形式的教育,虽然目前还没有任何这种渠道的广播课程。

遗憾的是,鉴于目前软件领域大量项目失败,所有教育渠道加起来可能也不足以提高软件工程和管理能力水平以使其达到完全专业的地位。

4.11 软件管理和技术类主题课程

表 4-7 列出了笔者推荐的软件行业管理类和技术类相关主题的全部课程,包括从顶级高管研讨会到详细技术课程。该课程集合所面向的是拥有大量软件员工的大型企业。假定这些课程的讲师都是该领域的顶尖专家。

该课程列表并非一成不变,随着新主题和新技术的出现,这些课程至少每年更新一次,也可能更加频繁。

表 4-7 面向高管、管理和技术人员的软件课程
Capers Jones (版权 © 2007 ~ 2009 by Capers Jones. 版权所有)

高管课程	天数	顺序	高管课程	天数	顺序
全球金融与经济	1.00	1	2008 年软件安全问题	1.00	4
软件开发经济学	1.00	2	软件架构趋势	1.00	5
软件维护经济学	1.00	3	软件外包经济学	1.00	6

(续)

高管课程	天数	顺序	项目管理课程	天数	顺序
离岸外包的优点和缺点	1.00	7	挣值度量	1.00	22
软件知识产权保护	0.50	8	评估与员工关系	1.00	23
《萨班斯-奥克斯利法案》 ^① 合规性	1.00	9	项目管理知识体系	2.00	24
软件诉讼规避	0.50	10	合计	34.00	
最佳实践案例研究	0.50	11	软件开发课程	天数	顺序
软件风险规避	0.50	12	软件架构原理	1.00	1
软件失效案例研究	0.50	13	软件结构化开发	2.00	2
CMM 概述	0.25	14	易错 (Error-Prone) 模块避免	1.00	3
六西格玛经济学	0.25	15	软件需求分析	1.00	4
病毒与间谍软件概述	0.50	16	软件变更控制	1.00	5
合计	11.50		2008 年软件安全问题	1.00	6
项目管理课程	天数	顺序	黑客和病毒防护	2.00	7
软件项目规划	2.00	1	联合应用设计 (JAD)	1.00	8
软件度量与指标	2.00	2	代码静态分析	1.00	9
软件成本估算: 手工	1.00	3	正式设计审查	2.00	10
软件成本估算: 自动	2.00	4	正式代码审查	2.00	11
软件质量和缺陷评估	1.00	5	测试用例设计和构建	2.00	12
软件安全规划	1.00	6	测试覆盖率分析	1.00	13
软件里程碑跟踪	1.00	7	不良修复 (Bad Fix) 注入降低	1.00	14
软件成本跟踪	1.00	8	软件缺陷报告与跟踪	1.00	15
软件缺陷跟踪	1.00	9	迭代和螺旋开发	1.00	16
软件变更控制	1.00	10	敏捷开发方法	2.00	17
功能点: 经理	0.50	11	Scrum 使用	1.00	18
软件审查: 项目经理	0.50	12	面向对象设计	2.00	19
软件测试: 项目经理	2.00	13	面向对象开发	2.00	20
六西格玛: 项目经理	2.00	14	Web 应用设计与开发	2.00	21
TSP/PSP 原则: 经理	1.00	15	极限编程 (XP) 方法	2.00	22
平衡计分卡原理	1.00	16	使用 TSP/PSP 开发	2.00	23
软件重用原则	1.00	17	数据库开发原则	2.00	24
软件风险管理	1.00	18	功能点: 开发	1.00	25
能力成熟度模型 (CMM)	2.00	19	可重用代码设计	2.00	26
六西格玛: 绿带	3.00	20	可重用代码开发	2.00	27
六西格玛: 黑带	3.00	21	合计	41.00	

① 《萨班斯-奥克斯利法案》，其全称为《2002 年公众公司会计改革和投资者保护法案》，由参议院银行委员会主席萨班斯 (Paul Sarbanes) 和众议院金融服务委员会 (Committee on Financial Services) 主席奥克斯利 (Mike Oxley) 联合提出。该法案针对在美国上市的公司提出，旨在监管上市公司“遵守证券法律以提高公司披露的准确性和可靠性，从而保护投资者及其他目的”。——译者注

(续)

软件维护课程	天数	顺序	软件质量保证课程	天数	顺序
遗留应用改造原则	1.00	1	使用 TSP/PSP 进行缺陷去除	2.00	10
易错模块去除	2.00	2	软件静态分析	2.00	11
复杂性分析与降低	1.00	3	软件测试用例设计	2.00	12
安全缺陷的识别和消除	2.00	4	软件测试库管理	1.00	13
不良修复注入降低	1.00	5	不良修复注入降低	1.00	14
软件缺陷报告与分析	0.50	6	测试用例冲突和错误	1.00	15
软件变更控制	1.00	7	功能点: 质量度量	1.00	16
软件配置控制	1.00	8	ISO 9000-9004 质量标准	1.00	17
软件维护 workflow	1.00	9	CMM 概述	1.00	18
多应用批量更新	1.00	10	软件重用质量保证	1.00	19
COTS ^① 包维护	1.00	11	COTS 和 ERP 质量保证	1.00	20
ERP 应用维护	1.00	12	六西格玛: 绿带	3.00	21
软件代码静态分析	1.00	13	六西格玛: 黑带	3.00	22
业务规则的数据挖掘	1.00	14	合计	35.00	
软件回归测试	2.00	15	软件测试课程		
软件测试库控制	2.00	16	软件测试用例设计	2.00	1
测试用例冲突和错误	2.00	17	软件测试库控制	2.00	2
死代码隔离	1.00	18	软件安全测试概述	2.00	3
功能点: 维护	0.50	19	软件测试进度估算	1.00	4
逆向工程	1.00	20	软件缺陷评估	1.00	5
再工程	1.00	21	缺陷去除效率度量	1.00	6
软件重构	0.50	22	静态分析和测试	1.00	7
可重用代码维护	1.00	23	软件测试覆盖率分析	1.00	8
面向对象维护	1.00	24	减少不良修复注入	1.00	9
敏捷与极限代码维护	1.00	25	识别易错模块	2.00	10
合计	27.50		数据库测试设计	1.00	11
软件质量保证课程			不正确测试用例去除	1.00	12
易错模块分析	2.00	1	单元测试基础知识	1.00	13
软件缺陷评估	1.00	2	回归测试基础知识	1.00	14
软件缺陷去除效率	1.00	3	组件测试基础知识	1.00	15
软件缺陷跟踪	1.00	4	压力测试基础知识	1.00	16
软件设计审查	2.00	5	病毒测试基础知识	2.00	17
软件代码审查	2.00	6	实验室测试基础知识	1.00	18
软件测试审查	2.00	7	系统测试基础知识	2.00	19
代码静态分析	2.00	8	外部 Beta 测试基础知识	1.00	20
2008 年软件安全与质量	2.00	9	测试用例冲突与错误	1.00	21

① COTS 即 Commercial off the shelf, 商用现成品或技术。——译者注

(续)

软件测试课程	天数	顺序	软件项目办公室课程	天数	顺序
Web 应用测试	1.00	22	软件失败案例研究	1.00	18
COTS 应用包测试	1.00	23	最佳实践案例研究	1.00	19
ERP 应用测试	1.00	24	软件里程碑跟踪	2.00	20
功能点：测试度量	1.00	25	软件成本跟踪	2.00	21
可重用功能测试	1.00	26	软件缺陷跟踪	2.00	22
合计	32.00		供应链评估	2.00	23
软件项目办公室课程	天数	顺序	平衡计分卡度量	1.00	24
软件项目规划	3.00	1	挣值度量	1.00	25
软件成本估算	3.00	2	软件六西格玛度量	1.00	26
软件缺陷评估	2.00	3	TSP/PSP 度量	1.00	27
软件功能点分析	3.00	4	软件价值分析	1.00	28
软件架构问题	1.00	5	ISO 9000-9004 质量标准	1.00	29
软件安全问题	1.00	6	代码行与功能点逆火分析 ^①	1.00	30
软件变更管理	2.00	7	度量指标转换	1.00	31
软件配置控制	2.00	8	合计	49.00	
软件审查概述	1.00	9	总课程	天数	科目数
软件测试概述	1.00	10	软件高管教育	11.50	16
软件度量与指标	2.00	11	软件项目管理教育	34.00	24
软件外包合同开发	1.00	12	软件开发	41.00	27
COTS 采购 (Acquisition)	1.00	13	软件维护	27.50	25
ERP 采购与部署	2.00	14	软件质量保证	35.00	22
《萨班斯-奥克斯利法案》合规性	2.00	15	软件测试	32.00	26
多公司外包项目	2.00	16	软件项目办公室	49.00	31
软件风险管理	2.00	17	合计	230.00	171.00

参考文献

- Boehm, Barry, Dr. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981.
- Curtis, Bill, Dr. *Human Factors in Software Development*. Washington, DC: IEEE Computer Society, 1985.
- Jones, Capers. *Applied Software Measurement*, Third Edition. New York: McGraw-Hill, 2008.
- Jones, Capers. *Estimating Software Costs*, Second Edition. New York: McGraw-Hill, 2007. (中文版《软件项目估计》(第2版), 电子工业出版社, 2008年3月出版)
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Boston: Addison Wesley Longman, 2000. (中文版《软件评估、基准测试与最佳实践》, 机械工业出版社, 2003年4月出版)

① Backfiring LOC to Function Points, 代码行与功能点逆火分析, 是一种将代码行 (Lines of Code) 数据与等效功能点数据之间直接进行数学转换的方法。详细信息请参见 1995 年 11 月 IEEE 论文《Backfiring: converting lines of code to function points》。——译者注

- Humphrey, Watts S. *PSP—A Self-Improvement Process for Software Engineers*. Upper Saddle River, NJ: Pearson Education, 2005.
- Humphrey, Watts S. *TSP—Leading a Development Team*. Upper Saddle River, NJ: Pearson Education, 2006.(中文版《TSP—领导开发团队》,人民邮电出版社,2007年1月出版)
- Kawasaki, Guy. *Selling the Dream*. Collins Business, 1992.
- Pressman, Roger. *Software Engineering—A Practitioner's Approach*, Sixth Edition. New York: McGraw-Hill, 2005. (中文版《软件工程:实践者的研究方法》(第6版),机械工业出版社,2007年1月出版。最新中文版为原书第7版中译版,机械工业出版社2011年5月出版)
- Weinberg, Dr. Gerald. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971. (最新中文版《程序开发心理学》(银周年纪念版),电子工业出版社2010年3月出版)
- Yourdon, Ed. *Decline and Fall of the American Programmer*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall, 1992.
- Yourdon, Ed. *Rise and Resurrection of the American Programmer*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall, 1996.

软件团队的组织和专业化

5.1 引言

不同于几乎任何其他技术或工程领域，软件开发严重依赖于人的智力、人力投入以及团队组织。从软件项目启动的那天直到可能 30 年后该软件退役之时，在软件开发、功能增强、维护和客户支持的每一个环节，人的因素都至关重要。

软件需求来源于人们对软件应用功能特征的讨论。软件架构依赖于人类专家的知识。软件设计则是建立在人们对客观世界认识的基础之上，依靠工具而不是智力来增强对某些事务的处理，但其不能智能地解决任何问题。

软件代码是由程序员一行一行编写的定制化人工制品，是现代人造产品中人力投入最大的。（创作雕塑或者构建特殊产品，比如一艘 12 米长的赛艇或者定制家具，需要熟练工匠类似数量的手工投入，但这些都不是广泛应用于成千上万公司的主流产品。）

尽管软件行业存在许多自动化静态分析方法和某些形式的自动化测试，但发现软件 bug 或安全漏洞还需要人类智慧。超过 95% 的软件应用项目及几乎所有规模大于 1000 个功能点的软件应用，仍然广泛使用手工审查，并手工创建测试计划和测试用例。遗憾的是，即便如此，软件质量和安全仍然是软件的薄弱环节之一。

随着全球经济陷入衰退，定制软件开发的高成本及边际质量和安全的状况正越来越引起软件高管们的重视。全球经济衰退很可能会强有力地促使人们从定制软件开发向构建标准的可重用组件转移。全球经济衰退为设计更高质量、更高安全性的软件提供了动力，使软件产业朝着高水平质量控制和安全控制自动化方向前进。

尽管存在“软件是所有人造产品中人力投入最高的”这一事实，但当前的软件文献中却并没有很好地涉及软件团队组织结构这一主题。

在诸如结对编程、小型自组织团队、敏捷团队、集中办公团队、矩阵式与层级式组织、项目办公室以及其他一些组织的价值上，有一些充满趣闻轶事的报告，但这些报告缺乏量化结果，因而很难找到经验数据以显示相同类型软件应用在不同类型组织之间的对比结果。

人们可以从国际软件基准组织（ISBSG）发布的一系列报告和数据中获得大量与团队相关的信息。例如，可以从这个组织得到规模在 1 ~ 20 人之间团队的生产力和平均软件规模数据。也可以得到更大型团队的数据。不过，那些超过 500 人的特大型团队除外，因为这

类团队很少向任何基准研究组织报告他们团队的基准数据。

5.2 量化组织结果

本章将用一种非同寻常的方式来讨论组织结构问题，深入探讨各种组织结构及其规模，并提供相关量化数据信息说明如下几个重要主题。

1. 就项目经理、软件工程师和专家而言的典型团队人员编制。
2. 特定规模和类型组织所能处理软件项目的最大规模。
3. 特定规模和类型组织所能处理软件项目的平均规模。
4. 特定规模和类型组织观察到的平均生产率。
5. 特定规模和类型组织观察到的平均开发周期。
6. 特定规模和类型组织观察到的平均质量水平。
7. 软件项目的人口数据，或者不同组织结构在软件行业的大概使用情况。
8. 各种不同组织结构下所配置的各种专家的人口信息。

当然，不同规模和种类的组织结构之间会有很多重叠。本章旨在缩小组织结构不确定性的范围，阐述对各种不同规模和类型的软件项目使用什么样的组织形式最适合。

就典型部门大小而言，本章所讨论的组织从只有一个人的软件项目到跨国家、跨学科的大型团队都有涉及，那些大型团队拥有的团队成员可能有 1000 名甚至更多。

本书中各种组织结构的观察结果来自于笔者对一些组织多年的实地考察。某些笔者实地考察过的组织包括：安泰保险、苹果公司、AT&T、波音、Computer Aid Incorporated (CAI)、Electronic Data Systems (EDS)、埃克森美孚、Fidelity、福特汽车、通用电子、哈特福德保险 (Hartford Insurance)、IBM、微软、美国国家航空航天局 (NASA)、美国国家安全局 (NSA)、索尼、德州仪器 (TI)、美国海军及其他超过 100 家组织。

组织结构是软件项目成功的一个重要方面，目前仍然需要对组织结构这一领域进行大量的实证研究。

5.3 割裂的信息技术和系统软件世界

诸如银行、保险等行业的许多大中型公司都有信息技术 (IT) 部门。虽然这些公司有自己的组织结构矛盾和各种问题，但像苹果、思科、谷歌、IBM、Intel、Lockheed、微软、摩托罗拉、Oracle、Raytheon、SAP 等公司的组织结构矛盾和问题则更加突出，这些公司通常开发系统软件和嵌入式软件，同时也开发 IT 软件。

在大多数构建 IT 软件和系统软件的公司，这两个组织往往截然不同。通常来讲，IT 部门报告给首席信息官 (Chief Information Officer, CIO)。而系统软件部门则通常汇报给首席技术官 (Chief Technology Officer, CTO)。

CIO 和 CTO 通常是平级的，所以谁的权力也不比另一个大多少。这两个完全不同的软件组织在培训、工具、方法学甚至编程语言方面共享甚少。他们往往位于不同的大楼内，甚至不同的国家。

因为系统软件部门往往是利润中心，而 IT 部门则往往是成本中心，所以这两个部门之间经常就会产生冲突，甚至相互厌恶。

系统软件部门往往为公司带来收入，而 IT 部门则往往没有收入。系统软件部门的福利待遇水平常常高于 IT 部门这一事实使得彼此的冲突愈加严重。

虽然 IT 和系统软件之间存在明显不同，但彼此也有众多相似之处。随着全球经济衰退的加剧以及很多公司寻求资金节约之道，在 IT 部门和系统软件部门之间共享信息似乎受益良多。

两个部门均需要在安全、质量保证、软件测试以及软件可重用性等方面获得培训。这两个部门往往处于不同的业务周期之中，因此有可能系统软件部门增长而 IT 部门规模萎缩，或者相反。经济衰退时期，在这两个部门之间设立协调职位就变得非常有价值。

同样，两个部门在都用得到的特定技能资源上进行彼此分享也颇具价值。例如，优秀的技术作家长期缺乏，技术交流不能同时服务于 IT 部门和系统软件部门则毫无道理。

诸如软件测试、数据库管理及质量保证等其他团体也可同时服务于系统软件和 IT 组织。

因此，只要经济衰退持续降低公司销售业绩并引发裁员，采用系统软件和 IT 这两个独立部门的组织将会发现其中的优势而考虑彼此进行合作。

尽管在质量方面系统软件通常都优于 IT 软件，但这两种软件经常都不具有最优秀的软件质量。诸如 PSP、TSP、正式审查、静态分析、自动化测试以及其他成熟的质量控制方法均可被 IT 组织和系统软件组织采用，这些都能够简化培训并允许从业人员更轻松容易地从—个部门转移到另一个部门。

5.4 集中办公与分布式开发

软件工程文献支持这样一种假设：在相同复杂程度的情况下，集中在同一地点工作的开发团队要比相同规模分散于不同城市或国家的分布式团队更加高效。

实际上，由笔者所做的一项涉及（比如操作系统和通信系统等）大型应用软件的研究表明，与那些相同规模始终集中在一个地点的团队相比，开发相同应用软件的分布式团队，每增加一个城市，生产率下降约 5%。

上述研究量化了从一个城市到另一个城市的出差成本。对于一个由分别位于欧洲 6 个城市和美国 1 个城市的分布式团队联合开发的大型通信应用软件，机票和出差的实际成本高于编程或编码的成本。该应用的开发团队整体规模为 250 人，至少 30 名软件工程师或专家每周需要从一个国家出差到另一个国家，而这种情况已持续 3 年多了。

遗憾的是，集中办公对软件开发有益这一事实表明，软件工程仍是门手艺（craft）或者一种艺术形式（art form）而不是一个真正的工程领域。对于大多数工程产品，比如飞机、汽车以及游轮，许多组件和子组件都是由许多分散在世界各地的分包商承建的。虽然这些分散制造的部件必须在同一地点进行最后的组装，但它们不需要在同一建筑中构建使其颇具成本效益。

软件工程在设计和开发上缺乏足够的精确度以允许远程开发部分组件之后集中交付进行最后建造。当然,很多软件项目确实包含了外包和远程开发团队,但当前的研究结果表明其生产率要比集中办公的开发团队低一些。

笔者有关远程开发的研究是在20世纪80年代完成的,当时,还没有Web和互联网使得跨地域通信像如今这样轻松。

如今,电话会议、网上研讨会、Wiki、Skype以及其他高带宽通信方式普遍使用。未来,更加成熟先进的通信方式也将投入使用。

设想有3个彼此相差8小时时差的分散开发团队,每天工作结束时,在大型应用上的工作可以从一个时区的团队传递给另一个时区的团队。依靠切换工作给位于彼此8小时时差的3个不同国家的团队,可以实现24小时不间断地进行开发。对于一个持续低迷多年的大型软件应用开发周期,对比位于同一个地点的开发团队,这种形式的分布式开发可能缩短约60%的开发周期。

要想使这种设想成为现实,显然,软件行业需要成为一个工程行业而不是一门手艺(craft)或一种艺术形式(art form),这样分散的开发团队才可以协调一致地工作而不破坏彼此的工作成果。尤其是,架构、设计与编码实践必须被所有位于3个不同地点的团队充分理解和彼此分享。

未来可能会出现一天24小时都可用的虚拟开发环境。在这种环境中,开发团队的虚拟化可以通过自己的真实照片或图像进行“面对面”地沟通。还可以使用Skype或类似工具进行现场通话,也包括电子邮件和用于远程设计和代码审查等活动的各种专业工具。

此外,成套的设计工具和项目规划工具也将在虚拟环境中得到使用,这样,技术和商业讨论都可以在这个环境中进行而不再需要昂贵的差旅费用。实际上,一个包含了所有团队的状态、bug报告、问题、进度状况和其他可创建项目材料的虚拟作战室甚至比今天集中于一地的开发组织更加有效。

这里的想法是,让3个相隔数千英里的独立团队以与集中办公开发团队相同的效率运作。这对于软件质量也同样令人向往,甚至比今天做得更好。当然,24小时不间断开发,项目进度将比现今通常情况下要快得多。

截至2009年,虚拟环境技术还没有达到大型系统开发所需要的有效成熟程度。但是,随着经济衰退的延续,那些能够降低成本(尤其是差旅费)的方法将会频繁地重新评估。

类似于那些在汽车、飞机和大型邮轮建造等工程实践领域广泛使用的准时制(Just-in-Time)软件工程实践可能将是涉及分布式开发、更加成熟有效的软件工程形式。

在这种情况下,需要支持构建可重用组件的标准架构。这些组件可能或者是已有存货,或者由专业厂商开发,其地理位置可能在这个星球上的任何地方。

这种实践的基本思路是,不同于定制设计和定制编码,标准架构和标准设计将允许使用标准可重用组件进行软件构建。

当然,这个想法涉及很多迄今还不完全存在的软件工程技术主题,比如部件列表、标准接口、质量和安全的认证协议以及支持可重用构建的架构方法。

截至2009年,应用软件定制开发的费用范围在每功能点1000~3000美元之间。软件

维护和改进的费用范围在每功能点每年 100 ~ 500 美元之间,一直没有改变。这些高成本使软件跻身于人类曾经创造的最昂贵商业“机器”之列。

随着经济衰退的延续,很明显,需要仔细分析高成本的定制软件开发活动并开发出更具经济效益的方法。理论上,可以通过由地理上分散的团队来装配经过认证的可重用组件,来显著降低开发成本、缩短开发周期。

软件工程师们的业务目标之一是将软件开发成本降低至每个功能点低于 100 美元,年度软件维护和改进费用低于每个功能点 50 美元。

自然而然的业务目标推论就是,将 10 000 个功能点的软件应用开发周期从目前的平均超过 36 个月降低到 12 个月或者更短。

潜在缺陷应该从目前的每功能点平均多于 5.0 个缺陷降低到少于 2.5 个缺陷。同时,平均缺陷去除效率水平应当从目前的少于 85% 提升到大于 95%,理想情况下高于 97%。

集中办公的开发方式并不能使软件项目成本、进度和质量如此大幅度地变化,但远程开发、虚拟开发环境和标准可重用组件的组合,将很好地把软件工程变成一个真正的工程领域,并显著降低软件开发和维护成本。

5.5 软件专家组织面临的挑战

你可能会认为,那些标题含有“软件工程”字眼的书籍的主要读者是那些从事新软件开发工作的软件工程师。虽然这样的软件工程师是该类书籍的主要读者,但他们实际上还不到大型公司里所有软件从业人员的三分之一。

当今世界,很多公司里从事遗留应用增强和修改工作的人员比新应用开发的人员要多得多。一些公司的软件测试人员与从事传统软件工程的人员一样多,有时甚至更多。

其他一些软件职业和软件工程师一样,对于软件项目取得圆满成功同样重要。这些关键员工同软件工程师一起并肩工作,没有他们的工作,大多数软件项目都无法完成。软件项目雇用的这些专业技能人员包括架构师、业务分析师、数据库管理员、测试专家、技术文档作家、质量保证专家和安全专家。

正如第 4 章和其他章节讨论过的,由于不一致的职位职称、不一致的职位描述以及诸如可能包括多达 20 种不同工作职位和职业的“技术人员”等抽象职称名称的使用,软件专业化主题的研究困难重重。

本章会涉及一个重要问题。出现如此多样化的专业技能和职位,它们对软件项目都非常必要,那么管理软件团队组织结构的最佳方式是什么?

这些专家应该嵌入到层次式组织结构中吗?他们应该是矩阵式软件组织结构的一部分,直接汇报给自己的行政管理线领导并“虚线”汇报给项目经理吗?他们应该是小型自组织团队的一部分吗?

截至 2009 年,有关该如何对各种软件专家进行组织和管理主题仍然显得含混不清、充满争论,而且很少有基于经验研究的可靠数据。一些确凿的事实已为人们所知,然而:

1. 软件质量保证人员需要得到切实保护免受威胁,以保持对软件质量的真实客观看法,

并诚实报告软件问题。因而，QA 组织需要单独从开发组织中分离出来，向上逐级汇报，直到主管质量的高级副总裁。

2. 由于软件维护和缺陷修复工作迥异于新软件开发，拥有大量遗留软件应用投资组合的大型公司应当考虑设立独立的维护部门以专司缺陷修复工作。

3. 如果所在的部门或团队主要由软件工程师组成，一些专家如技术文档作家等将很少有机会获得晋升或工作丰富化（job enrichment）。因而，一个独立的技术出版物组织将会给这些专家提供更好的职业发展机会。

对软件专家来说，基本的问题是，他们是应该基于技能为单位和其他分享相同技能和职称的人组织在一起，还是嵌入到他们实际使用这些技能的职能部门？

基于技能为单位组织和管理专家^⑤的好处是，可以提供给这些专家更多的职业机会和更好的教育机会。同样，当出现损害或丧失工作能力的情况下，基于技能的组织通常可以迅速指派他人接管相关工作。

同很多其他各种技能专家一道加入某个更大职能组织^⑥的优势在于，对该职能组织的工作，可以立即获得所需要的专家。

通常，如果有大量某种类型的专家（如技术文档作家、软件测试员、质量保证等），基于技能的组织似乎更具优势。而对于那些稀缺技能（如安全、架构等），可能没有足够多相同职位的人手来组建一个基于技能的组织。

本章会讨论涉及软件相关核心专家组织结构的各种替代方法。目前，总共有多达 120 种软件相关的专家，但对于很多类型的专家，即使在相当大型的公司，也可能只有一到两个此类专家雇员。

本章集中讨论关键专业，他们的工作对大公司里大型软件应用项目的成功至关重要。假定在一个相当大型的公司的软件组织里，总共有 1000 名软件从业人员。在这 1000 名从业者里，最有可能雇用多少种不同的专家及多少特定个体工作者呢？对于上述问题，项目成功最重要的专家是什么？表 5-1 识别出了这总数 1000 名软件从业者中某些重要专家及其近似的组成情况。

由表 5-1 可见，软件工程师不必亲自从事所有工作。在现代世界里，为开发和维护应用软件，需要各种其他技能。实际上，截至 2009 年，经济衰退继续延续，形势持续严峻，虽然经济衰退会降低软件人员的绝对数量，但软件专家的数量和种类仍在持续增长。

⑤ 这里所讲是指各种以专业技能为基础而组建的部门或团队，如软件测试、质量保证、软件设计、国际化、开发、维护、客户支持等部门或团队。其典型特征是全部人员均由相同专业技能的专家组成，从事相同或相似的工作。具有这些部门的公司，多数以矩阵式架构、员工多线汇报、项目制运作为主。——译者注

⑥ 这里所讲的职能组织或部门，与 PMI PMBOK 所讲的“职能部门”略有不同。这里是指以产品线或公司业务为主，由具有各种技能的专家和员工共同组成的部门。这种部门多数为层级式架构、员工单线汇报，结构相对简单。——译者注

表 5-1 总数 1000 名软件人员中软件专家的分布情况

		数量	百分比
1	软件维护专家	315	31.50%
2	软件开发工程师	275	27.50%
3	软件测试专家	125	12.50%
4	一线经理	120	12.00%
5	质量保证专家	25	2.50%
6	技术文档作家	23	2.30%
7	客户支持专家	20	2.00%
8	配置控制专家	15	1.50%
9	二线经理	9	0.90%
10	业务分析师	8	0.80%
11	范围经理	7	0.70%
12	行政支持	7	0.70%
13	项目库管理员 (project librarian)	5	0.50%
14	项目规划专家	5	0.50%
15	软件架构师	4	0.40%
16	用户接口专家	4	0.40%
17	成本估算专家	3	0.30%
18	度量 / 指标专家	3	0.30%
19	数据库管理专家	3	0.30%
20	本地化 (Nationalization) 专家	3	0.30%
21	图形艺术家	3	0.30%
22	软件性能专家	3	0.30%
23	软件安全专家	3	0.30%
24	软件集成专家	3	0.30%
25	软件加密专家	2	0.20%
26	可重用性专家	2	0.20%
27	测试库控制专家	2	0.20%
28	项目风险专家	1	0.10%
29	标准专家	1	0.10%
30	价值分析专家	1	0.10%
	总软件雇员	1000	100.00%

5.6 由小到大的软件组织结构

笔者观察到的软件组织规模，从最低一个人的单人团队，多到 30 人或更多人的跨行业团队都有。

由于历史原因，软件团队的“平均”规模通常是 8 个团队成员加一名经理或者团队组长。然而，更小或更大的团队也相当常见。

本节将会从小到大、从单个人的项目开始，逐一探讨软件项目组织结构的大小规模和性质。

5.6.1 单人软件项目

成立单人 (one-person) 项目最常见的企业目的是为了进行遗留应用的维护和小规模改进。对于新应用软件开发,在企业环境中,构建 Web 网站也是个典型的单人活动。

然而,实际上,相当多只有一个人的软件公司在从事开发小型商业软件包的工作,比如 iPhone 应用、共享软件、自由软件、计算机游戏以及其他小型应用软件。事实上,相当多的创新性新软件和产品的创意都来自于这些只有一个人的公司。

人口数据 因为小型软件维护项目比较常见,所以在美国,每天可能有接近 250 000 个单人项目正在进行,其中绝大部分是遗留应用的维护和功能增强。

就生产小型应用软件的单人软件公司而言,据笔者估计,截至 2009 年,美国可能存在超过 10 000 个这样的公司。这是个令人惊讶、富有成果的创新之源,并在开源软件、自由软件和共享软件领域广泛存在。

项目规模 由单人项目完成的新应用软件的平均规模大小是 50 个功能点,最大规模低于 1000 个功能点。对于软件维护和缺陷修复类工作,平均规模少于 1 个功能点,少数项目至多会高至 5 个功能点。对于遗留应用的功能增强,平均规模大小为每个新添加功能约 5~10 个功能点,少数会高至 15 个功能点。

生产率 单人工作的生产率通常相当不错,每人月可高达 30 个功能点。一个需要解释的地方是,如果单人开发团队也不得不自己编写用户手册、提供客户支持,那么其生产率将削减近一半。

另一个需要说明的是,许多单人公司都是在家里办公的。因此,诸如流感、新生婴儿等意外事件,或者其他一些正常的家庭活动,如婚丧嫁娶,都能对手头的工作产生显著影响。

第三个需要特别警告的是,单人软件项目对特定个体的技能水平和工作经验非常敏感。可控实验证明,对于编码或缺陷去除等任务,最好结果和最坏结果可能有 10 倍的显著差异。也就是说,相当多迁移到单人项目职位的工作者,都是具有高端竞争力和优秀业绩水准的人。

进度 单人软件维护和功能增强项目的开发日程常常从一天到一周范围内不等。对于由一个人完成的新应用开发,开发日程范围通常在 2~6 月之间。

质量 单人完成的应用软件质量水平都还不错。潜在缺陷可达到每个功能点约 2.5 个缺陷,而缺陷去除效率约 90%。因此,一个有 25 个功能点的小 iPhone 应用可能总共含有约 60 个缺陷,而发布时仅有 6 个缺陷。

专业化 你可能会认为,既然很显然诸如测试和文档等全部特殊技能都能在单个个体身上找到,单人项目应该是那些通才 (generalist) 的天地。然而,仔细考察那些单人项目后,令人惊讶的是,很多这样的项目都是由那些既不是软件工程师也不是程序员的人实施完成的。

对于嵌入式软件或系统软件,很多单人软件项目都是由电子工程师、通信工程师、自动化工程师或者其他类型的工程师负责完成的。即使商业软件,一些单人项目也可能是由会编程的会计师、律师、业务分析师或其他领域专家完成的。这就是如此众多数量的发明和新想法从这些小公司和单人项目中不断涌现的原因之一。

警告与注意事项 开发和维护单人项目最主要的警告是，在生病或者丧失工作能力的时候，缺乏任何后备资源。如果项目上唯一的一个人发生了什么意外事情，项目工作将彻底停止。

第二个警告是，如果开发软件的人是领域专家（比如会计师、业务分析师、统计学家等），其正在公司里构建一个为个人使用的应用软件。在这个雇员离开公司的时候，可能会涉及该软件所有权的法律问题。

第三个警告是，在由知识工作者开发的软件含有错误或者对公司或其客户确实造成某些损害的情况下，可能产生软件可靠性问题。

结论 单人项目对于遗留应用软件小的功能提高、更新和维护变更非常常见且相当有效。

尽管单人开发项目的规模必须相当小，但同时，数量惊人的创新和优秀思想均来源于这些才华横溢的个体实践者。

5.6.2 软件开发与维护的结对编程

结对编程的想法是指两个软件开发者共享一个计算机轮流编程，当一个开发者编程时另一个作为观察者来检查编程者的工作。两个角色以固定时间间隔频繁来回切换，比如每30分钟到一小时切换一次。编程的团队角色称为“驾驶者”，另一个团队角色称为“导航者”或“观察者”。

截至2009年，结对编程的效果并不明朗。一些研究表明结对编程缺陷较少，而另一些还宣称开发进度也得到了改善。

但是，所有结对编程的实验规模都相当小，且关注点相当狭窄。例如，没有任何已知的结对编程缺陷与使用静态分析和自动化测试的个体程序员编程结果之间的对比研究，也没有任何最优秀的个体与平庸的结对编程者平均水平之间的对比研究，反之亦然。

也没有任何已知研究将结对编程的软件质量和那些已证明有效的质量方法的质量结果进行对比，比如设计和代码的正式审查，这些方法具有近乎半个世纪的经验数据可用，且利用他人的服务来寻找软件缺陷。

虽然实验表明，很多结对编程的开发周期较短，但没有任何实验可以证明让2个人做正常由一个人完成的工作可以节省多少开发工作量或成本。

对于结对编程，要降低成本，进度必须加快至少50%。然而，相关实验和最近收集到的数据表明进度加快只有大概15%~30%，与一个人单独做同样的工作相比，这将增加开发成本超过50%。

结对编程狂热者们断言，更好的质量将会弥补较高的开发付出和成本，但这种说法并没有得到那些包括静态分析、自动化测试、正式审查和其他成熟缺陷去除方法研究的支持。两个开发者使用手工缺陷去除方法进行开发要比使用同样手工缺陷去除方法的单个开发者开发的软件产品中缺陷更少这一事实饶有趣味，但不足以令人信服。

对于可重用组件开发，结对编程可能是个有趣且非常实用的方法，它要求非常高的质量和可靠性，而开发所付出的努力和成本相对并不重要。还有，Watts Humphrey的团队软件

过程(TSP)对可重用组件开发也是很好的选择,且这方面的可用历史数据比结对编程要多。

主观上讲,结对编程概念似乎对那些经历过它的人是个愉快的体验。通常认为,让另一个同事参与到更加复杂的算法和代码结构中来对软件开发是有利的。

随着2009年经济衰退继续扩大,裁员越来越多,由于很多公司将软件员工人数缩减到最小水平且不再负担额外开销,很多公司很可能将不再有人使用结对编程。

大多数结对编程的文献书籍都涉及在同一个办公室的集中办公,偶尔也会提到合作伙伴在另一个城市或国家的远程结对编程。

结对编程是个有趣的协作形式,而相互协作对于规模大于100个功能点的应用软件总是必须的。

在测试驱动开发(TDD)的环境中,结对编程的一个有意思的变化是,结对的一个人负责编写测试用例,而另一个人则负责编写程序代码,然后再切换角色。

另一个已成功使用结对编程的领域是软件维护和缺陷修复。有一个软件维护外包公司曾依照城市警察局的模式^①来组织他们的软件维护团队。这样做的原因是,缺陷的出现具有很大随机性,当有新缺陷报告,尤其是新的高严重性缺陷被报告的时候,总是需要有可用人手来处理这些缺陷。

在软件维护的警察模式中,一个分发者和几对彼此搭档一起工作的维护程序员,就像警探彼此搭档一起工作一样。

在缺陷分析期间,让两个团队成员并肩工作可以加速找到已报告缺陷的来源。让两个团队成员合伙工作在缺陷修复上也可以加快缺陷修复速度,降低不良修复注入。(从历史数据看,大约7%的缺陷修复会意外地在修复代码中引入新的缺陷,这被称为不良修复注入。)

实际上,结对编程在缺陷修复和软件维护活动中的使用看起来可能是使用结对最有效的方式。

人口数据 由于结对编程是一种实验中的方法,因此该方法并未广泛推广。随着经济衰退的延续,结对编程可能会变得更少。据笔者估计,截至2009年,在美国可能仍然活跃着500~1000对结对编程小组。

项目规模大小 由结对编程所完成的新应用开发的平均规模大约为75个功能点,最大规模少于100个功能点。对于软件维护和缺陷修复工作,平均规模少于1个功能点。对于遗留应用的功能增强,每个新加功能的平均规模为大约5~10个功能点。

生产率 结对编程的生产率经常在每人月16~20个功能点之间,比由单人完成的不同项目少30%。

结对编程软件项目受特定个体的技能和工作经验的影响非常大。正如以前提到过的,可控实验表明,在这种研究的个体参与者完成的诸如编码和缺陷去除等任务上,最好结果和最差结果可能有高达10倍的显著差异。

某些软件从业人员的心理研究指出了一种内向性倾向,这种倾向可能导致某些软件工

① 美国警察通常2人搭档组成小组行使职责,无论上街巡逻、调查刑事案件、维护交通,还是处理邻里纠纷,通常都由2人搭档小组来执行,其形式类似于这里所讲的“结对”。——译者注

程师对结对编程概念感觉不舒服。结对编程文献确实也说明了这种社交满意度的情况。

进度 使用结对编程的软件维护和功能增强项目的开发进度范围经常在一天到一周之间。使用结对编程的新软件开发项目，进度范围通常在2~6个月之间。对于同样数目的功能点，结对编程项目的进度通常比单人项目短大约10%~30%。

质量 结对编程应用软件的质量水平还不错。潜在缺陷达到每个功能点约为2.5个缺陷，而缺陷去除效率约为93%。因此，一个有25个功能点的小iPhone应用可能有总计60个缺陷，而发布时仍会有4个缺陷。这比那些使用手工缺陷去除和测试方法的个体开发者好15%。然而，当前没有任何结对编程与使用了自动化静态分析和自动化测试的个体程序员所做工作之间的比较数据。

专业化 到目前为止，关于结对编程环境中专业化角色的研究并不多。但是，已有一些关于结对编程工作中任务分布的有趣报道。例如，结对中的一个人负责编写测试用例而另一个人则负责编码，或者一个人负责编写用户故事而另一个人则负责编写程序代码。

到目前为止，还没有关注那些在相同软件应用具有显著不同背景团队的结对编程研究。也就是，一个软件工程师与一个电子工程师或者一个自动化工程师组对儿；一个软件工程师与一个医生组对儿，等等。不同学科的配对儿可能是个值得尝试的事情。

警告与注意事项 如果确实如上述所说，在结对编程方法成为主流软件开发方法之前，需要做更多额外的实验研究。这些实验需要包含更多成熟的质量控制方法，并需要与最优秀的个体程序员进行对比。在当前这种严重经济衰退期间，结对编程的高成本使其不太可能拥有过多追随者。

结论 关于结对编程，几乎没有足够的经验数据用以得出可靠的结论。结对编程的实验和奇闻轶事通常是正面积极的，但是到目前为止，这些实验只涵盖了很少的变量，且忽略很多重要主题，比如静态分析、自动化测试、审查及其他质量因素的作用。随着全球经济衰退的不断延续和深化，由于软件组织裁员和机构缩减，结对编程很可能逐渐从人们的视线中淡出。

5.6.3 自组织敏捷团队

近几年来，随着敏捷开发运动获得更多的拥护者，小型自组织团队的概念也获得了更多追随者。自组织团队的概念不同于让团队成员报告给经理或者正式团队组长，这种团队的每个成员将转换到他们感觉最舒服最匹配他们技能的团队角色。

在自组织团队里，每个成员都是最终交付物的直接贡献者。在设有一名经理的普通部门，经理通常并不是代码等要交付给终端用户的最终交付物的直接贡献者。因此，自组织团队应该比相同规模的普通部门稍微更有效率，因为他们额外有一名工作者。

在美国商业界，普通部门每名经理平均有大约8名下属。给经理汇报的雇员数称为“管理跨度”^④。(大型公司比如IBM实际观察到的管理跨度范围从每名经理最低2名到最高30

④ 管理学上，管理跨度，又称控制幅度、管理幅度、管理扇面，是指在一个组织结构中，管理人员所能直接管理或控制的部属数目。这个数目有一定限度，当超过这个限度时，管理效率就会随之下降。不同管理者能力有所不同，其管理跨度亦不同。每个管理者必须认真考虑自己的最佳管理跨度。——译者注

名雇员不等。)

对于自组织团队,名义规模范围是“7加减2”^①。然而,为真正匹配任何给定大小的软件项目,团队规模范围需要扩大到低至2名、高达12名成员。

软件领域存在的一个显著历史问题是,将应用软件进行分解以匹配现有的组织结构,而不是基于基础架构将软件应用分解为对应的逻辑块。

这样的实际效果是,将一个大型软件应用划分为多个功能块,这样每块可由一个8人部门或团队进行开发,无论这种划分是否匹配软件应用的体系结构。

在敏捷环境中,用户代表成为团队一员,能为团队提供未来所需的输入,也能提供根据团队已完成的应用程序片段运行情况而得出的体验报告。用户代表具有特殊作用,一般不会做任何代码开发工作,但某些测试用例可能由加入团队的客户代表创建。显然,用户将在用户故事、用例和他们所需功能的非正式描述方面提供有价值的输入。

在理论上,自组织团队是跨职能团队,每个人都根据需要对每个可交付物做出贡献。然而,这对那些偏离自己主要能力范围的人并不特别有效。技术文档作家可能不是个优秀的程序员。而能成为优秀技术文档作家的人又寥寥无几。所以,往往只有每个团队成员根据自己的特长为项目贡献力量时才会取得最好的效果。

然而,在每个人都是(或者没有任何人是)熟练工种的情况下,所有人都可参与项目工作。有效测试用例创建也许就是整个团队稀缺技能的一个例子。处理代码安全也是个很少有团队成员熟练的领域,如果这是个严重问题,那就必须引入外部专家以支持团队工作。

自组织团队的另一个方面是每日状态会议的使用,这个会议被称为Scrum每日例会,Scrum是一个源自橄榄球运动的词汇。典型地,Scrum每日例会非常简短且只涉及3个关键问题:(1)自上次会议之后都完成了哪些工作,(2)今天和下次会议之间计划要完成的工作,以及(3)遇到了什么问题或障碍。

(每日例会并不是Scrum唯一分享信息的方式。打电话、发邮件以及非正式面对面交谈天天发生。在多个团队之间根据需要也可能有较大的会议。)

自组织团队一个备受争议的角色是Scrum Master。名义上,Scrum Master是整个项目协调员的一种形式,负责为其所管理的团队成员设定工作期望。也就是,Scrum Master是整个自组织团队的教练。这个角色意味着Scrum Master运用自己的个人魅力和领导才能对整个团队施加强大影响力。

人口数据 因为敏捷开发近几年快速增长,小型敏捷团队的数量也在增长。截至2009年,笔者估计单在美国就有大约35 000个小型自组织团队,整体上雇用了大约250 000名软件工程师和其他专业的从业者。

项目规模 由7个人组成的自组织团队完成的新应用开发项目平均规模是大约1500个功能点,其最大规模可达3000个功能点。(超过3000个功能点,则需要使用“团队的团队”(teams of teams)这种形式。)自组织团队很少用在软件维护或缺陷修复工作中,因为一个缺陷的平均规模小于1个功能点,一个人就能搞定。对于遗留应用软件的功能增强项目,

① 即5~9名成员。——译者注

自组织团队可能主要适用于规模在 150 ~ 500 个功能点范围内的项目。对于 5 ~ 10 个功能点的小型功能增强项目，可能会由一个人编写程序为主，辅以其他角色如测试、技术文档作家和集成专家。

虽然有些方法可以用于扩展小型团队为包括“团队的团队”在内的大型团队，但自组织团队的扩展仍是个问题。实际上，整个敏捷哲学似乎更适用于少于 2500 个功能点的应用。超过 10 000 个功能点的大型系统软件曾试图使用敏捷或自组织团队的例子极少。

生产率 工作在 1500 个功能点规模项目上的自组织团队的生产率通常在每人月 15 个功能点左右。在团队里有显著的专业知识时他们的生产率可高达 20 个功能点，而对于罕见或复杂项目其生产率则可能低到每人月只有 10 个功能点。

单个 Sprint 的生产率通常较高，但这其实并不重要，因为这样的 Sprint 不包括所有组件的最终集成、整个软件应用的系统测试和最终的用户文档。

自组织团队项目往往尽量缩小团队单个个体的业绩差别，这可能有助于新手的快速成长。但是，如果某些团队成员的个人业绩差别超过 2 : 1，那些业绩最好的团队成员将对业绩最差成员的工作充满不满。

进度 对于典型的 1500 个功能点项目，由自组织团队负责完成的新应用开发周期常常在 9 ~ 18 个月之间，对整个软件，平均需要 12 个月。

然而，敏捷方法将整个应用程序划分为一组每个都可单独开发的程序分段。这些分段被称为 Sprint，典型地，其规模大小可以在一到三个月内完成开发的规模。对于一个 1500 个功能点的应用，可能需要 5 个 Sprint，每个 Sprint 完成 300 个功能点。每个 Sprint 持续 2.5 个月的时间。

质量 自组织团队达到的软件质量还不错，但通常没有达到团队软件过程（TSP）等方法的软件质量水平。在这些方法里，软件质量是一个核心问题。典型潜在缺陷可达每功能点 4.5 个缺陷，缺陷去除效率约为 92%。

因此，由自组织敏捷团队开发的具有 1500 个功能点的应用，总共会有 6750 个缺陷，发布时仍会有 540 个缺陷没有被去除，其中约 80 个为严重缺陷。

然而，如果使用诸如自动化静态分析和自动化测试等工具，缺陷去除效率可达到 97%。在这种情况下，软件发布时只有 200 个遗留缺陷没有被去除，其中 25 个为严重缺陷。

专业化 迄今为止，只有少数关于自组织团队专业化角色的研究。事实上，一些自组织团队的狂热者更鼓励在自组织团队使用通才（generalist）。他们往往将专业化看作类似于流水线上的工作。然而，通才经常在培训和经验方面明显不足。对自组织团队有所帮助的专家种类包括安全专家、测试专家、质量保证专家、数据库专家、用户接口专家、网络专家、性能专家以及技术文档作家。

警告与注意事项 关于自组织团队的主要警告是，该类组织缺失标准的、易于理解的团队结构，使出现权利斗争和破坏性人际冲突的机会大大增加。

第二个警告是，将敏捷方法从小应用软件向上扩展到具有多个地点、多个开发团队的大型系统软件时，已被证明这是件既复杂又困难的事情。

第三个警告是，与敏捷相关的软件质量度量实践十分糟糕，以及许多自组织团队给予

了敏捷方法狂热崇拜的光环而不是将之视为严谨的工程学科。不能有效度量生产力或质量,或者不能使用标准度量指标来报告基准数据,都是敏捷方法的严重缺陷。

结论 自组织敏捷团队的文献和证据好坏参半,且经常含糊不清。敏捷方法蓬勃发展的头五年,自组织团队赢得了各种主观性文章的大量赞赏。

然而另一方面,从2007年开始,由于自组织团队令人迷惑的角色分派、团队内部破坏性的权力斗争以及彻底的项目失败,越来越多地出现大量文章和报告不断质疑自组织团队,甚至有文章建议应该取消自组织团队。

这是软件行业的典型发展模式。某些富有感召力的行业大牛最初倡导某种新的开发方法,然后该方法开始获得大量文章和书籍的积极评价,即使通常没有任何经验数据或量化结果加以佐证。

几年之后,新开发方法的各种问题开始引起人们注意,越来越多使用这种新开发方法的应用项目走向失败或者无法取得成功。部分是因为培训不足,但主要原因是,几乎没有软件开发方法在开始大规模部署之前经过受控条件下的全面分析和使用评估。糟糕的质量度量实践方法和基准研究数据的缺失,也是使软件方法评估进展缓慢的一个长期问题。

遗憾的是,自组织团队起源于敏捷开发实践的大环境。敏捷方法无论是在生产力度量还是软件质量度量方面均比较差,且几乎没有有效的基准研究数据。当度量敏捷项目时,往往使用特殊的度量指标如故事点数或用例点数,这些指标既不规范也缺乏经验数据和基准研究数据的收集。

5.6.4 团队软件过程(TSP)团队

团队软件过程(TSP)的概念是 Watts Humphrey 基于他在 IBM 的工作经历而开发的,他也是软件工程研究所(SEI)的软件能力成熟度模型(CMM)的创始人。

TSP 概念涉及成功完成软件开发所需要的各种角色和职责。但 TSP 是构建在个体技能和责任基础之上的,所以需要与个体软件过程(PSP)一起考虑。通常情况下,软件工程师和专家首先学习 PSP,然后转向 TSP。

由于 Watts Humphrey 的 IBM 工作背景及能力成熟度模型(CMM)的背景,TSP 方法与能力成熟度集成模型(CMMI)相一致,并满足 CMMI 5 级的很多标准,而 CMMI 5 级是 CMMI 体系的最高等级。

TSP 团队是自组织团队,他们与同为自组织团队的敏捷团队表面上十分相像。然而,敏捷团队往往基于分配给团队的任何成员的技能 and 爱好而采用不同自由形式的团队结构。

另一方面,TSP 团队是建立在一个明确团队成员角色和职责的坚实基础之上的,从一个项目到另一个项目保持不变。因此,TSP 团队中,团队成员是根据那些对于软件项目成功非常必要的特定技能标准而挑选出来的。缺乏所需技能的雇员可能无法成为 TSP 团队成员,除非提供足够的培训。

同样,接受 PSP 的先期培训也是加入 TSP 团队的强制条件。其他种类的培训,比如估算、审查以及测试等,也可作为先导(precursor)而提供给要加入 TSP 团队的雇员。

敏捷团队和 TSP 团队另一个有趣的区别在于这两种方法的起源。敏捷方法由那些主要

关注 1500 个或更少功能点、相对小型的 IT 应用软件的实践者创始的。而 TSP 方法则由那些主要关注 10 000 个或更多功能点、大型系统软件应用的实践者所创始的。

出发点的不同导致了技能和专业化的一些差异。因为小型应用使用更少专家，敏捷团队经常是由根据需要能够处理设计、编码、测试甚至文档等工作的通才构成的。

因为 TSP 团队经常参与大型应用开发，他们往往使用诸如配置控制、集成、测试等方面的专家。

敏捷团队和 TSP 团队均非常关注软件质量，但他们通常以极为与众不同的风格追求软件质量。一些敏捷方法基于测试驱动开发，也就是说在开始编写程序代码之前先编写测试用例。这种方法相当有效。然而，敏捷方法往往回避正式审查，并且在记录缺陷和度量软件质量方面不够严格。

而对于 TSP 团队，关键可交付物的正式审查及正式测试是其质量控制的主要组成部分。另一个主要区别是，在度量从需求阶段第一天开始到最终交付期间所遇到的每一个单个缺陷上 TSP 非常严格，而敏捷项目中的缺陷度量则有些稀稀落落，且测试之前通常不会度量缺陷。

敏捷和 TSP 团队均使用自动化缺陷跟踪工具，均使用诸如静态分析、自动化测试和自动化测试库控制等方法。

敏捷和 TSP 的其他一些差异不一定会影响软件项目的成果，但这些差异确实会影响人们对这些成果的理解。敏捷方法在生产力和质量度量方面比较松懈，而 TSP 团队则在任务小时数、挣值、缺陷数量和其他许多可量化因素方面非常严格。

因此，当软件项目完成时，敏捷项目只有些含混不清或缺乏说服力的数据以表明项目的生产力和软件质量结果。而与之相反，TSP 项目则有大量可靠的量化数据。

尽管层次式结构更为常见，TSP 团队用于层级式组织结构和矩阵式组织结构均可。Watts Humphrey 报告说，TSP 可被用于许多不同类型的软件，其中包括国防软件、民政管理应用软件、IT 应用软件、诸如 Oracle 和 Adobe 等公司的商业软件，甚至还被一些计算机游戏公司所使用，这些地方已经证明了 TSP 在消除令人讨厌的缺陷方面很得力。

人口数据 TSP 广泛使用在那些雇用了总数超过 1000 ~ 50 000 名软件从业人员的大型组织。由于 TSP 和 CMMI 之间的协同作用，它也被军事和国防软件组织广泛采用。这些大型机构通常有大量特殊技能的人才和数以百计在同时进行的软件项目。

据笔者估计，美国有大概 500 家公司在使用 TSP。虽然某些公司的使用可能是实验性质的，但由于该方法的成功，实际使用量增长相当迅速。2009 年，美国使用 TSP 的软件人员数量可能达到 125 000 人。

项目规模 具有 8 名成员和 1 名经理的 TSP 团队所完成的新开发项目平均规模大约是 2000 个功能点。然而，TSP 组织可以扩展到任意规模，故甚至超过 100 000 个功能点的大型系统也可以由多个 TSP 团队通过协同工作来完成。对于具有多个 TSP 团队的大型应用软件，诸如测试、配置控制及集成等很多专家团队也被用以支持一般的开发团队。

当多个 TSP 团队试图相互合作以完成项目时，另一个需要注意的地方是，当超过数十个 TSP 团队同时参与到项目中时，可能需要某些类型的项目办公室，以整体规划和协调多

个团队之间的协作。

生产率 负责 2000 个功能点规模项目上的 TSP 团队的生产率通常在每人月 14 ~ 18 个功能点之间。对于那些团队中有显著专业知识的应用软件开发, TSP 团队的生产率可高达每人月 22 个功能点, 而对于那些不寻常和复杂的项目, TSP 团队的生产率可能会降低到少于每人月 10 个功能点。随着应用程序规模越来越大, TSP 团队的生产率往往反比于应用规模而下降。

进度 负责 2000 个功能点的项目上、拥有 8 名成员的 TSP 团队, 其新应用软件开发的开发周期通常在约 12 ~ 20 个月范围内, 整个应用的平均周期为 14 个月。

质量 TSP 团队的质量水平异乎寻常的优秀。TSP 团队的平均潜在缺陷为每个功能点约 4.0 个缺陷, 而缺陷去除效率率则可达到 97%。交付缺陷只有每功能点平均约 0.12 个。

因此, 单个 TSP 团队开发完成的 2000 个功能点的应用, 总共有 8000 个缺陷, 其中 240 个缺陷到发布时仍未得以去除, 而这之中只有约 25 个是严重缺陷。

然而, 如果额外使用预测试审查、诸如自动化静态分析和自动化测试等工具, 缺陷去除效率可高达 99%。在这种情况下, 发布时只遗留了约 80 个缺陷, 其中只有 8 个是严重缺陷, 每个功能点只有 0.004 个缺陷。

通常, 随着应用规模的增加, 潜在缺陷也会增加, 而缺陷去除效率水平则会下降。有意思的是, 在 TSP 方法中, 这条规则似乎并不成立。很多规模较大的 TSP 应用软件可以实现与小型应用软件或多或少类似的质量水平。

TSP 方法的另一个令人惊讶的发现是, 随着应用规模的增长, 生产力水平似乎并没有明显下降。一般地, 生产力会随着应用规模的增长而下降, 但 Watts Humphrey 报告说, 在大范围的规模应用上 TSP 团队并没有出现明显的生产力下降。这种说法需要更多的研究来加以证实, 如果确实如此, TSP 方法将成为众多软件开发方法中唯一不随软件规模增长而生产力下降的开发方法。

专业化 TSP 设想了各种各样的专家。大多数 TSP 团队都会拥有大量的主题专家, 如软件架构、测试、安全、数据库设计和其他许多方面的专家。

有趣的是, TSP 方法并不推荐软件质量保证 (SQA) 成为标准 TSP 团队的一部分。这是因为, TSP 团队自己本身对质量控制的要求非常严格以至于根本不需要 SQA。

在那些由 SQA 负责收集质量数据的公司里, TSP 团队可以根据需要提供这些数据, 但这些数据将由团队自己的成员负责收集和提供而不是交给 SQA 人员或者分配给项目的其他人员。

警告与注意事项 关于 TSP 组织和项目的主要注意事项是, 虽然 TSP 度量许多重要的主题, 但它并不使用标准的度量指标, 比如功能点。TSP 使用任务小时数 (Task Hours) 或多或少有些独特, 但很难将任务小时数和标准指标直接进行对比。

另一个注意事项是, 很少有 TSP 项目已向任何正式的软件基准研究组织 (比如国际软件基准组织 ISBSG) 提交过基准数据。结果, 在没有进行复杂数据转换的情况下, 几乎不可能将 TSP 方法与其他方法进行对比。

从技术上讲, 使用几种新的高速功能点方法计算功能点总数是可行的。实际上, 无论

是对新应用还是遗留应用，量化功能点现在只需要几分钟时间。所以，使用功能点方法报告 TSP 团队的软件质量和生产力并不是特别困难。

将任务小时数转换为正常的工作周或工作月信息有点儿麻烦，但毫无疑问，这些数据可以使用算法或者某些基于规则的专家系统进行转换。

采用高速功能点方法并向一个或多个基准研究组织比如 ISBSG 提交项目基准结果数据，无论对敏捷项目还是 TSP 项目，都是极为有利的。

结论 TSP 方法通常能够成功实现很高质量水平的应用软件，且很少有项目失败。因此，值得人们深入研究。

从那些涉及项目失败或应用软件从未成功运行的诉讼案例中所做的观察可知，TSP 还没有任何项目失败以致闹上法庭而告终的事情发生。这种情况可能会随着 TSP 应用数目的极大增长而有所变化。

TSP 强调团队管理人员和技术人员的能力，也强调有效的质量控制和变更管理控制。有效的估算和仔细的进度跟踪也是 TSP 项目的典型特征。TSP 人员在使用该方法之前要经过严格培训，以及常备有经验丰富的指导者可用等诸多事实，解释了为什么 TSP 方法很少被误用。

例如，对于敏捷开发，有十数个或者更多的关于开发活动该如何进行的变种方法，但他们仍然使用“敏捷”作为统称。而 TSP 活动则被更为认真严格地加以定义和使用，所以，当笔者访问多家公司的多个 TSP 团队时，他们提到了相同的方法执行相同的活动。

由于注重质量，TSP 将是标准可重用组件构建方法的一个不错选择。TSP 方法也是那些糟糕质量可能会导致严重后果的“危险”应用（如医疗系统应用、武器系统应用、财务应用等）的一个不错选择。

5.6.5 层级式组织结构的传统部门

层级式组织的概念是这个地球上分配角色和职责的最古老方法。“层级结构”这个词的词汇来自希腊语，意思是“祭司统治”。但这个概念本身比希腊还古老，在埃及、苏美尔以及大多数其他古代文明中均可发现它的身影。

很多宗教组织都是以层级结构方式组织的，就像军事组织一样。有些企业也是层级结构的，如果它们是由私人拥有的。具有大量股东的上市公司通常是半层级式结构的。在这种公司里，运营部门向上一级一级汇报，直到总裁或首席执行官（CEO）。然而，首席执行官（CEO）报告给由股东选举出来的董事会主席，所以，上市公司的最高层并非精确地是一个真正的层级结构。

在一个层级式组织里，各种不同规模的部门每个都有自己由上级主管部门任命的正式领导或经理。虽然这个任命的主管部门经常也是下一个更高级别组织的领导，实际的任命权通常是由该层级结构组织的最高层委托下来的。一旦被任命，每一个领导都向其相同行政管理体系的更高层领导汇报。

虽然各级被任命的领导或经理们有权发出命令或者指导自己所负责部门的工作，他们也必须服从来自上级的指示。进度报告反向汇报给更高级的主管部门。

在商业企业层级结构中，低层经理们通常由上一层的经理任命。但对于高管职位，如副总裁，则可能由最高层管理人员组成的一个委员会负责任命。这样做的目的是，至少在理论上，确保这种层级组织的高级管理人员有能力胜任所在职位。然而，最近出现的金融业动荡和不断扩大的全球经济衰退表明，太多公司的高层管理人员往往是整个层级式组织结构中最薄弱环节。

需要注意的是，一个组织的实际层级结构和它的权力结构可能并不一致。例如，在中世纪的日本，天皇是正式的政府层级结构的最顶端，但实际统治权却被授予了一个以称为“幕府将军”的指挥官为首的军事组织。只有天皇才可以任命将军，但具体任命受军事领导者主宰，而天皇则几乎没有任何军事或政治权力。

层级式结构组织的一个为时已久的问题是，如果位于金字塔顶端的领导者个性软弱或者能力不强，整个层级组织就会面临着失败的危险。对于层级式政府，软弱的领导人可能会导致革命或丧失自己的领土给强势邻邦。

对于层级式商业组织，最高领导人的无能往往会导致市场份额丧失，甚至可能失败或者破产。实际上，近期从安然到雷曼等商业失败案例的分析确实表明，这些层级式组织的最高层管理人员确实没有必要的能力或洞察力来处理严重问题，或者理解问题到底是什么。

一个有趣的商业现象是，层级式企业的生存寿命近似地等于人类的寿命。极少有公司能活过100年的。随着全球经济衰退的不断延续和加深，大量公司很可能寿终正寝，但也有一些公司会乘机扩张并得以不断壮大。

层级式组织有两类雇员。一类是由普通工人或专家组成的，他们实际上从事着企业的真正工作。第二类是由企业工人或专家所报告给的各级经理和高层管理人员组成。当然，经理们也要汇报给更高层的经理们。

技术性工作和管理性工作的区别如此深刻地嵌入在层级式组织中，以至于产生了两个截然不同的职业路径：管理路线和技术路线。

当开始自己的职业生涯时，年轻雇员几乎总是以技术工人职位开始。对于软件行业，这意味着开始于软件工程师、程序员、系统分析师、技术文档作家等职位。工作几年之后，他们需要作出职业选择，要么获得提拔进入管理工作岗位，要么继续从事技术工作岗位。这种选择常常由个人性格与兴趣爱好来决定。许多人喜欢技术工作，从来不想进入管理领域。而另一些人则较享受团队活动的规划和协调工作，适合从事管理职业。

管理人员和技术工人在数量上极不平衡。在大多数公司，管理社区总数大约占总雇用人数的15%，而技术工人的数量则只占85%左右。由于经理通常不是公司生产过程的一部分，避免过多数量的经理和高管就显得非常重要。经理和高管太多会降低公司运营业绩。这已在商业和军事组织中引起关注。

有趣的是，当高到一定级别时，技术员工和管理人员的薪酬待遇可以大致相同。例如，在大多数公司，高级别技术员工的薪酬可以达到三线经理的水平。但是，在企业的最高层，仍然存在严重不平衡的薪酬待遇。

许多公司的CEO和一些执行副总裁拥有价值数百万美元的薪酬奖金。实际上，在一些公司里，某些高管的薪酬奖金超过普通员工薪酬的250倍。随着全球经济衰退的加深，这

些巨额的高管薪酬奖金正在受到股东和政府监管部门的质疑。

另一个开始受到质疑的是管理跨度，或者汇报给经理的技术员工数目。因某些不明的历史原因，美国平均的部门人数是大概 8 名员工汇报给一名经理。观察到的人数范围，从每名经理带 2 名员工到 30 名员工都有。

假定每名技术经理平均带 8 名技术工人，那么全部员工的大约 12.5% 将会是一线经理。如果也包括高级别经理们，整体可占到 15% 左右。

分析大公司里经理们的考核成绩及研究针对经理们的投诉可知，在管理上合格有效的人少于 15%。实际上，只有 10%（或者更少）的人真正是合格、有效的。

那就是说，研究将管理跨度由目前的每名经理平均 8 技术员工提升到每名经理 12 名技术员工将很有意义。淘汰不合格经理并让他们回归到技术工作可能会提高公司整体效率，降低因管理不善引起的不稳定因素。

实践中很多经理可能会说，增加管理跨度会降低他们控制项目、理解他们部属实际工作状态的能力。然而，笔者在大公司（如 IBM）所做的经理们的时间和活动研究发现，软件经理们往往在与其他经理们的会议上花费更多时间，而不是与他们自己员工的会议和讨论上。实际上，一个可能的业务定律是“管理性会议的多少反比于管理跨度”。在特定项目上的经理越多，他们花费在与其他经理沟通而不是与他们自己员工交流的时间就越多。

这项研究的另一个富有争议的方面与项目的失败率、延误及其他事故有关联。对于有多名经理的大型项目，失败率似乎与参与到项目中的经理数目更加紧密相关，而非参与到项目中的软件工程师和技术文档作家等技术员工的数目。

虽然技术工人也经常需要完成他们自己的工作并与其他部门的同事好好相处，但管理上的付出往往因经理们的权力斗争和争论而大打折扣。

这需要更进一步的研究和验证。然而可以得出结论，增加管理跨度、减少管理人员数量往往会提高一个软件项目成功的概率。这对那些缺乏竞争力、错放在管理岗位上的经理尤其如此。

在许多拥有很多专家的层级式部门，同一个人可能既做开发又做维护。应该注意的是，如果同一个软件工程师既要负责开发同时又要负责维护工作，将非常难以精确评估其开发工作。这是因为，软件维护工作涉及的高严重性缺陷修复经常优先抢占软件开发任务的时间，因此打断软件开发的正常进度计划。

另一个主要主题是，在回顾技术员工的离职面谈时，提到了 2 个令人不安的事实：1）业绩考核得分最高的技术员工往往会大量离职；2）最常引述的离职理由是，“我不喜欢为糟糕的管理而工作。”

关于层级式组织管理的另一个有趣现象称为“彼得原理”（the Peter Principle）。彼得原理是由 Lawrence J. Peter 博士和 Raymond Hull 于 1968 年在其出版的同名书籍中提出的。本质上，彼得原理认为，在层级式组织中，员工和管理人员是根据他们的个人能力而不断获得提拔的，直到达到他们不再胜任的某一等级。结果是，很大比例的老员工和经理们从事着他们并不胜任的工作职位。

彼得原理可能令人觉得好笑（它首次也是以幽默书籍的形式出版的），但考虑到那些大

量被取消的项目,甚至更大数量的项目进度延误和成本超支,软件开发项目不能对彼得原理置若罔闻。

假设一个层级式软件组织的基本单位由8名员工汇报给1名经理组成,那么他们的头衔、角色和职责是什么呢?

通常,层级式组织模型多见于使用更多通才而较少专家的公司里。由于软件专业化经常随着公司规模而增加,其含义是,层级式组织结构被拥有较少技术员工的小型到中等规模公司广泛采用。通常,可以在那些雇用了5到50名软件从业人员的公司里找到层级式组织。

在这样的层级式组织结构中,主要工作头衔是程序员或软件工程师。这些从业人员既要负责开发工作,也负责软件维护工作。

然而,层级式组织在较大规模公司和那些确实雇用了专家的公司也可找到。这种情况下,一个8人部门可能是由5名软件工程师、2名测试员和1名技术文档作家及他们汇报给的1名经理这样的人员编制结构组成。

大企业有众多业务部门,如市场、销售、财务、人力资源、生产及可能还有研发部门。使用层级式组织原则,每个部门都有自己的软件组织,致力于构建只在特定业务部门使用的软件,即财务应用软件、生产制造支持应用软件,等等。

但是,当需要某种跨越所有业务部门的公司级或企业级应用时,又会发生什么事情呢?跨职能应用最终证明在传统的层级式或“大礼帽”^①式组织中是很难构建出来的。

为解决跨职能应用软件的上述问题,人们开发出了两种替代方案,矩阵式管理结构便是其中之一,将在本章下一部分详细讨论。第二个方案是企业资源规划(ERP)软件包,它是由诸如SAP和Oracle等大型软件厂商为解决跨职能商业应用而开发的。

正如下一部分将讨论的,矩阵式管理的组织类型经常被一些软件团体所使用,它们都具有广泛的专业和需要涉及跨职能应用以支持众多业务部门。

人口数据 软件世界里,层级式组织结构最常见于雇用5~50名软件从业人员的小公司。这些公司往往采用通才哲学而只有少数除了诸如网络管理和技术文档编写等一些技术性技能之外的专家。在采用通才哲学的环境中,上述由5~8名软件工程师汇报给1名经理的层级式组织同时负责软件开发、软件测试和软件维护等活动。

据笔者估计,在美国有大约10 000个这样的小公司。截至2009年,在美国,在这种层级式组织结构下工作的软件从业人员数量可能有250 000名。

层级式组织结构也常见于一些大公司,所以可能另有500 000名员工在层级式结构的大公司和政府机构中工作。

项目规模 8名员工和1名经理的层级式团队,其新应用开发的平均规模大小约为2000个功能点。然而,层级式组织的特征之一是他们可以在大型项目上进行合作,所以甚至超过100 000个功能点的大型应用软件,也可以由多个部门协同工作完成开发。

① 英语俚语,原意为“锅炉的烟囱或美式黑色大礼帽”,这里的含义类似于中文的“山头林立、各自为政”。——译者注

关于多个部门试图协作工作需要说明的是,当超过十数个部门同时参与项目时,可能需要使用某些类型的项目办公室,以整体规划和协调多个部门之间的协作。参与的某些部门需要处理集成、测试、配置控制、质量保证、技术文档编写以及其他方面的工作。

生产率 负责 2000 个功能点规模项目的层级式部门的生产率通常为每人月 12 个功能点左右。当团队有着显著的专业知识时,他们的生产率有时可高达每人月 20 个功能点。而对于不常见或复杂的项目,其生产率可能会降低到每人月 10 个功能点以下。生产率通常反比于应用规模,随着应用规模的不断增大而下降。

进度 负责 2000 个功能点的项目、拥有 8 名团队成员的单个层级式团队,其新应用开发的开发进度周期通常在 14 ~ 24 个月之间,整个应用平均为 18 个月。

质量 层级式部门的质量水平相当中等。潜在缺陷为每功能点约 5 个缺陷,而缺陷去除效率约为 85%。交付缺陷平均为每功能点 0.75 个缺陷。

因此,一个由单个层级式部门开发的 2000 个功能点的应用软件,总共有大约 10 000 个缺陷,软件发布时仍会含有其中 1500 个缺陷。当然,约 225 个为严重缺陷。

然而,如果使用预测试审查方法及诸如自动化静态分析和自动化测试等工具,那么缺陷去除效率可达 97%。这种情况下,发布时可能只有 300 个缺陷,而其中只有约 40 个是严重缺陷。

专业化 到目前为止,关于层级式软件组织结构中专业化角色的研究还比较少。鉴于通才们在培训和个人经验上普遍不足,大型应用软件开发仍需要某些类型的专家。对大型应用软件开发有所帮助的专家类别包括安全专家、测试专家、质量保证专家、数据库专家、用户接口专家、网络专家、软件性能专家和技术文档作家。

警告与注意事项 关于层级式组织结构的主要警告是,软件工作往往被人划分为匹配 8 人开发部门的能力,而不是依照应用软件自身的架构和设计来分段。结果,当应该由单个团队来负责完成时,大型系统中的某些较大功能却被武断地在两个或多个部门之间进行分割。

尽管特定部门内部的沟通交流比较容易和自发,但部门之间的沟通却由于经理们保护自己的领地而大大慢了下来。这样,对于由众多层级式部门参与的大型项目,主要在管理层,存在权力斗争和破坏性社交冲突的可能性很高。

结论 关于层级式组织结构的文献资料有趣但并不完整。许多关于层级式组织结构的文献都是由其替代形式组织的狂热者撰写的,这些替代形式的组织结构包括矩阵式管理、敏捷团队、结对编程、净室开发等。

自软件行业存在伊始,层级式组织就持续被软件应用开发所采用。虽然这个事实似乎表明其成功性,但不可否认的是,软件行业具有比任何其他行业项目失败、成本超支及进度落后的比率都要高的特点。截至 2009 年,层级式组织结构对软件项目成功或者失败的真实影响仍然并不明确。

诸如开发方法、雇员技能水平及管理技能等其他因素往往与组织的结构因素交织在一起,使得识别组织本身对软件项目的影响变得极为困难。

5.6.6 矩阵式组织结构的传统部门

矩阵式管理的历史比软件开发自身的历史还要年轻。最早关于矩阵式管理的文献似乎起始于20世纪60年代末期,当时矩阵式管理被NASA用于处理复杂太空计划相关的跨功能项目。

矩阵式管理的思想很快从NASA扩展到了民用领域,最终被软件组织所采用以处理专业化和跨职能应用软件。

在传统层级式组织结构的一个具体业务部门,各种各样的软件从业人员汇报给同一个经理。普通技术雇员可能是各种通才,或者部门里也有各种专家。比如软件工程师、测试员和技术文档作家。如果一个特定的业务组织里包含10个软件部门,每个软件部门都会有许多软件工程师、测试员、技术文档作家,等等。

与之相比,在一个矩阵式组织中,各种职业团体和专家汇报给一个相同技能或职业的经理。这样,所有的技术文档作家可能汇报给一个技术出版物组织;所有的软件工程师都在一个软件工程组织中;而所有的测试员可能都在一个测试服务组织中;以此类推。

通过把各种知识工作者整合到一个以技能为基础的组织中,与把专家零星分散在多个层级部门里相比,可以使他们的工作更加丰富多彩并提供更多的职业发展机会。

在矩阵式组织中,当各种项目需要专家时,会将所需专家分配给该项目,整个项目期间专家临时汇报给项目经理。当然,这引出了员工同时为两个老板工作的棘手局面。

经理之一(通常是技术经理)拥有专家雇员的业绩考核和薪水评定权限,而另一个经理(通常是项目经理)则使用专家雇员的服务以完成项目。项目经理可以向技术经理提供关于专家雇员工作业绩的信息。

拥有对员工业绩考核和薪酬评定权限的经理被称为拥有“实线”汇报权限。而只是因特定任务或特殊项目而借用该专家的经理被称为拥有“虚线”汇报权限。这两个词反映了组织结构图的绘制方式。

一个有趣的现象是,矩阵式管理思想太过崭新,以至于SAP、Oracle及其他一些ERP应用软件的早期版本并不支持“虚线”汇报或者矩阵式管理结构。截至2009年,所有的ERP软件包都已实现对矩阵式组织结构图的支持。

大约在2009年,关于矩阵式管理的文献资料在爱好者和反对者之间出现了极其剧烈的两极分化。大约一半的书籍和文章将矩阵式管理视为一个重大的商业成就。而另一半书籍和文章则将矩阵式管理视为令人费解、充满破坏性和重大的商业不利。

在Google上搜索短语“矩阵式管理的失败”,会返回315 000个引用,而搜索短语“矩阵式管理的成功”则会返回327 000个引用。由此可见,对矩阵式管理的看法强烈地两极分化但几乎又彼此平分秋色。

近些年来,出现了3种矩阵式管理形式:弱矩阵、强矩阵和平衡矩阵。

初始的矩阵式管理形式现在已被归类为弱矩阵。在这种组织形式中,雇员主要汇报给技能经理,根据需要外借给项目经理。项目经理没有考核雇员业绩和评估薪水的权力,因此依靠自愿协作的方式完成项目工作。如果项目经理和技能经理在资源配置上出现冲突,项目经理缺乏获取他们项目所需技能资源的权力。

由于弱矩阵管理组织被证明麻烦重重,因而很快出现了强矩阵管理形式。在强矩阵管理组织中,专家可能仍然汇报给技能经理,但一旦被分配给一个项目,根据项目需要就会取得该专家资源的最高优先级。实际上,在整个项目实施期间,专家有可能被正式地分配给项目经理,接受项目经理的业绩考核和薪水评估。

在平衡矩阵管理组织中,名义上在技能经理和项目经理之间平等地共享职责和权力。虽然这听起来像是一个好主意,但实践证明它很难实现。结果,截至2009年,强矩阵似乎仍然是占主导地位的矩阵式管理形式。

人口数据 软件世界里,矩阵式组织最常见于雇用软件从业人员总数在1000到50000名之间的大公司里。这些大公司往往有大量的专业化技能的专家和数以百计同时进行中的项目。

据笔者估计,美国有大约250个以矩阵式管理组织为主的大公司。截至2009年,在美国,工作在这种矩阵式管理结构下的软件从业人数达到100万人。

项目规模 有8名成员和1名经理的矩阵式团队负责完成的新应用开发,其平均规模大约为2000个功能点。然而,矩阵式管理组织可以扩展到任意规模大小,所以,即使规模超过100000个功能点的大型系统应用也可以由多个矩阵式组织通过协同工作来完成。

当多个矩阵式团队试图合作完成项目时,需要说明的是,当超过十数个不同团队同时参与项目时,可能需要使用某些类型的项目办公室,以整体规划和协调多个团队之间的协作。

对于真正超过25000个功能点的大型应用软件,某些部门可能完全由负责处理各种专业技术(比如集成、测试、配置控制、质量保证、技术文档编写和其他专业方面等)的专家组成。

生产率 在2000个功能点的项目上的矩阵式部门的生产率通常在每人月10个功能点左右。当团队有着显著的专业知识时,他们的生产率有时可高达每人月16个功能点,而对于不常见或复杂的项目,其生产率可能会降低到每人月6个功能点以下。生产率通常反比于应用规模,随着应用规模的不断增大而下降。

进度 负责一个2000个功能点的项目、由8名团队成员组成的单个矩阵式团队所负责新应用软件的开发生周期通常在约16~28个月之间。对整个应用软件其平均周期为18个月。

质量 矩阵式组织的质量水平通常比较中等。潜在缺陷可达每个功能点约5个缺陷,而缺陷去除效率约为85%。交付缺陷为平均约每功能点0.75个。除非诸如正式审查、静态分析、自动化测试和其他最顶尖方法等特殊方法也引入到项目中,否则矩阵式和层级式组织在软件质量方面具有相同水平。

所以,一个由单个矩阵式开发部门开发的2000个功能点的应用软件共有约10000个缺陷,产品发布时仍会有1500个缺陷没有被修复。当然,其中约225个为严重缺陷。

然而,如果使用预测试审查方法及自动化静态分析和自动化测试等工具,那么缺陷去除效率可达97%。这种情况下,发布时可能只有300个缺陷,而其中只有约40个是严重缺陷。

随着应用软件规模的增加,潜在缺陷数量也随之增加,而缺陷去除效率水平则随之下降。

专业化 矩阵式组织结构的主要目的是支持专业化。话虽如此,到目前为止,关于矩

阵式软件组织结构中专业种类的研究并不是特别多。截至 2009 年,人们仍然不清楚诸如各种规模应用软件所需的架构师数量、测试员数量及质量保证人员数量等方面的确切数据。

大型应用程序经常需要各种典型的专业种类。对大型应用软件开发有所帮助的专家类别包括安全专家、测试专家、质量保证专家、数据库专家、用户接口专家、网络专家、软件性能专家和技术文档作家等。

警告与注意事项 关于矩阵式组织的主要警告是技能经理和项目经理之间的政治纠纷。

另一个警告是,尽管很难评估,大约一半关于矩阵式组织结构的研究和文献断言,矩阵式组织结构方法弊大于利。然而,另一半研究和文献则说法相反,声称从矩阵式组织结构中获得了显著价值。但是,对于任何具有 50% 负面结果的方法,都需要认真考虑而不能盲目采用。

对矩阵式组织和层级式组织的一个共同的警告是,软件工作往往被人划分为匹配 8 人开发部门的能力,而不是依照应用软件自身的架构和设计来分段。结果,当应该由单个团队来负责完成时,大型系统中的某些较大功能却被武断地在两个或多个部门之间进行了分割。

尽管特定部门内部的技术交流比较容易和自发,但部门之间的沟通却由于经理们保护自己的领地而大大慢了下来。这样,对于由众多层级式部门或矩阵式组织参与的大型项目,主要在管理层,存在权力斗争和破坏性社交冲突的可能性很高。

结论 关于矩阵式组织的文献如此严重地两极分化,以至于很难找到任何共识。在一半文献称赞矩阵式组织而另一半因各种项目失败及软件灾难而指责矩阵式组织时,不太容易找到有说服力的可靠经验数据来证明任何一方的观点。

从那些涉及项目失败或应用软件从未成功运行的诉讼案例中所做的观察可知,层级式组织和矩阵式组织似乎相差不大。矩阵式组织和层级式组织所牵涉的诉讼案件数目基本相当。

真正造成差别的是管理人员和技术员工的能力以及强调效果的质量控制和变更管理控制。有效估算和严格的进度跟踪也具有一定区别,但这些因素没有任何一个直接与层级式或者矩阵式组织结构模式有关。

5.7 大型公司的专家组织

因为软件开发工程师并不是大公司和政府机构中唯一的或者最大的职业团体,那么什么类型的组织能够为最常见的职业群体提供最佳服务非常值得人们思考。

按雇用人数的粗略数字排序,主要的专家职位有:

1. 软件维护工程师
2. 软件测试人员
3. 业务分析师和系统分析师
4. 客户支持人员
5. 软件质量保证人员

6. 技术文档编写人员

7. 行政人员

8. 配置控制人员

9. 项目办公室人员

☐ 软件估算专家

☐ 软件项目规划专家

☐ 软件项目度量和指标专家

☐ 软件范围经理

☐ 过程改进专家

☐ 标准专家

很多其他种类的人员也从事技术性工作，比如网络管理、数据中心运营、工作站和个人电脑维修以及以运营而不是以软件为中心的其他活动。这些职业也很重要，但它们不在本书所讨论范围之内。

下面是对所选的专家团体组织的探讨。

5.7.1 软件维护组织

对于少于 50 名软件人员的小公司，软件维护和开发工作通常都由同一伙人来完成，没有独立的软件维护小组。在这种情况下，某些形式的客户支持工作也要小公司里的软件工程社区来负责。

然而，随着公司的不断壮大，软件维护工作往往会出现专业化分工。对于软件人员数目超过 500 人的公司，成立单独的软件维护团队已成为必然，而不再是特殊情况。

(注意：国际软件基准组织 (ISBSG) 拥有超过 400 个项目可用的软件维护基准数据，且每月都有新数据添加进来。更多信息，请参考 www.isbsg.org。)

将软件维护工作从软件开发中独立出来这种做法，既有批评者，也有支持者。

独立软件维护组织的批评者指出，将软件维护工作从软件开发中分离出来，需要额外人员来熟悉相同的应用软件，这可能会人为地增加总的人员编制。他们还断言，如果在同一个应用软件上的软件功能增强和缺陷修复任务由两伙不同的人同时负责完成，这两个任务可能会彼此干扰。

而独立软件维护组织的支持者们则声称，由于缺陷随机出现且数目相当大，它们会干扰正常的开发进度。如果同一个工程师既要负责为一个应用添加新功能又要负责修复发现的软件缺陷，当突然报告一个高严重性缺陷时，修复缺陷的工作将比开发工作具有更高的优先级。结果，开发进度就会下降，也许下降得非常严重甚至使应用程序的 ROI 为负。

虽然双方各有道理，但笔者的观察结果支持“独立软件维护组织对有大量软件需要维护的大型公司最有帮助”这一观点。

独立软件维护团队在发现和修复软件问题上比软件开发者有更高的生产率。同样，让独立软件维护变更团队进行开发将更具可预测性，可以提升开发生产力。

一些维护团队在负责缺陷修复的同时还负责处理应用软件的小改进。目前还没有关于

“小改进”的精确定义，但一个可行的定义是：由一个人在一周内能够完成的软件更新。这也将小改进的规模限制在 5 个或更少的功能点以内。

尽管缺陷修复和功能增强是最常见的两种软件维护形式，实际上，如表 5-2 所示，大型组织里共有至少 23 种不同形式的软件维护工作。

表 5-2 23 种软件维护工作

1	较大功能增强（超过 20 个功能点的新功能）	13	逆向工程（从代码中提取潜在的软件设计信息）
2	微小功能增强（少于 5 个功能点的新功能）	14	再工程（将遗留应用转换为现代形式）
3	维护（出于良好意愿而修复缺陷）	15	死代码去除（去除从来不用代码段）
4	保修（基于正式协议的缺陷修复）	16	冬眠应用清除（归档不再使用的软件）
5	客户支持（响应客户的电话呼叫或问题报告）	17	国际化与本地化（修改软件以国际化使用）
6	易错模块去除（清除非常棘手的代码段）	18	大规模改造（如欧元或千年虫问题修复）
7	强制性变更（要求的或法定的变更）	19	重构，或重新编程以改进软件清晰性
8	复杂度或结构化分析（控制流程图或复杂性度量指标）	20	退役（应用程序退出现役）
9	代码结构调整（减少程序循环和基本复杂度）	21	现场服务（派遣软件维护人员到客户现场提供维护服务）
10	优化（提高程序性能或吞吐量）	22	向软件厂商报告缺陷或错误
11	迁移（从一个平台向另一个平台移植软件）	23	安装接收自软件厂商的软件更新
12	转换（改变接口或文件结构）		

尽管这 23 种软件维护形式在很多方面都不尽相同，但它们都有一个值得深入讨论的共同特点：它们都涉及修改现有应用程序，而不是从头开始开发一个全新的应用软件。

对现存应用软件进行 23 种不同形式的修改，其原因各不相同。但是，经常发生的情况是不同形式的修改经常同时发生。例如在一个应用软件的某个版本里，功能增强和缺陷修复经常需要同时进行。

软件维护文献有许多关于软件维护任务的分类方法，比如以“适应性”、“纠正性”或“完美性”等来分类。这些分类方法似乎来自学术界，虽然没什么错，但却忽略了最基本的一点。整体上，软件维护工作只在经济上有 2 个真正重要的区别：

1. 客户承担费用的变更（功能增强）。
2. 构建该软件的公司承担费用的变更（缺陷修复）。

软件公司无论是使用软件维护的标准学术还是采用上述更加精细的 23 种形式来分类，将软件维护成本分别归类为“客户出资”和“自筹经费”这两类费用非常重要。

比如赛门铁克（Symantec）等一些公司会向客户的服务呼叫收取费用，甚至会因客户报告了软件缺陷而向客户收取费用。笔者认为，收取这些费用的做法很不专业，就像一个唯利是图的人为了赚钱而不惜牺牲质量控制。

这些软件修改活动有其一定的顺序或者模式。例如，逆向工程经常发生在再工程之前，而它们又经常一起进行几乎形影不离。对于大型应用软件及重要系统软件的发布版本，笔者曾观察到在同一个版本中出现上述维护形式中的 6 到 10 种。

近些年来,信息技术基础架构库(ITIL)已对软件维护、客户支持和服务管理产生了重大影响。ITIL是一个超过30本书和手册的超大型信息技术集合,它涵盖了服务管理、事故报告、变更团队、可靠性标准、服务协定和其他许多主题。本书写作的2009年,ITIL第三版正在实施中。

软件世界一个有趣的现象是,虽然ITIL已经成为IT服务公司里服务协议的主要驱动力,但它却几乎从来没有被像微软、赛门铁克等商业软件厂商用于它们与其客户之间的服务协议。实际上,上述软件厂商几乎总是要求用户在使用它们的软件之前阅读最终用户许可协议(EULA)里面的那些小字。这对软件厂商来说,是很有好处的。

当阅读这些协议时,每每看到那些声称无论如何软件厂商都没有任何责任的条款就令人不安,因为这些条款声称不能保证软件正常运行或者保证任何种类的质量水平。

出现这种单方面EULA协议的原因是,软件行业的软件质量控制实践太糟糕,如果因为它们的软件质量不佳造成的破坏而被起诉,即使大型软件厂商也要破产。

对很多IT组织和商业软件团体来说,很多功能是一起整合在一个更大的统称之下的:客户支持、软件维护(缺陷修复)、小功能增强(少于5个功能点)及有时还包括集成和配置控制。

此外,一些形式的软件维护还涉及非本公司自己开发的软件:

1. 那些采购自SAP、Oracle、微软等厂商的商业应用软件的维护。维护任务包括报告软件缺陷、安装新版本,可能还包括依据当地条件对软件进行定制、变更。
2. Firefox、Linux、Google等开源软件和自由软件的维护。这里,维护任务同样包括报告缺陷及安装新版本,以及根据需要进行定制。
3. 通过兼并或收购其他公司而加入到本公司投资组合中的软件的维护。这是一个饱含问题、危险重重且非常棘手的情况。这种情况下的软件维护任务可能非常复杂,涉及改造、大版本更新以及可能需要从一个数据库到另一个数据库进行数据迁移。

除了正常的软件维护任务,还有一种被称为革新(Renovation)的软件维护方式,它结合了缺陷修复和功能增强,对遗留应用软件进行彻底的、大范围的现代化改造。

软件革新包括易错模块的外科手术式清除、自动或手工程序结构再调整以降低复杂度、注释的修订或替换、死代码块去除以及可能还包括将遗留应用从旧的或过时的编程语言自动转换成较新的现代编程语言。

软件革新还包括数据挖掘以提取内嵌在遗留应用代码中而这些代码的规格说明书和书面描述遗漏了的业务规格和各种算法。静态分析和自动化测试工具也可以用于软件革新。同样,现在已经能够自动生成遗留应用的功能点总数了,这也可以成为软件革新的一部分。

目前所观察到的软件革新效果是,它延长了遗留应用软件至少10年的额外使用寿命。软件革新减少了遗留代码中的潜伏缺陷,因此减少了未来用于遗留应用改造的维护成本约每年50%。由此,客户支持成本也跟着降低了。

随着经济衰退的加深和延长,作为淘汰或重新开发遗留应用软件更具成本效益的替代方案,软件革新将变得越来越有价值。软件革新能够降低维护成本如此之多,完全可以用

这些节省下来的费用去开发新的应用软件。

如果一个公司确实计划改造其遗留应用软件,修复初始旧代码中毫无疑问会出现的那些长期慢性问题将非常合适。最明显的一个是去除通常在遗留应用软件中非常多的安全漏洞。第二个要做的是在软件革新期间使用正式审查、静态分析、自动化测试和诸如 TSP 等其他现代技术来改进软件质量。

团队软件过程(TSP)、来自 Google 的 Caja 安全架构以及比其他大多数语言都更为安全的 E 编程语言相互结合,可以用于改造那些处理财务或有价值专有数据的应用软件。

为预测软件维护工作相关的人员配备和工作量,根据对诸如 IBM、EDS、软件生产力研究所(SPR)及其他许多公司里软件维护团队的观察,人们开发了许多有用的经验法则。

软件维护任务量范围 一个自然年内单个软件维护程序员能够成功维护的软件规模。截至 2009 年,美国平均的软件维护任务量约为 1000 个功能点,变化范围在 350 个功能点到 5500 个功能点之间。影响软件维护任务量范围的因素包括软件维护团队的工作经验、代码复杂度、代码中潜伏缺陷的数量、代码中是否有易错模块以及诸如静态分析工具、数据挖掘工具和维护工作台等可用的工具套件等。软件维护任务量范围是预测所需软件维护程序员总数量的重要度量指标。

(对于大型应用软件,软件内部架构知识对于有效软件维护和修改都至关重要。因此,重大系统软件往往都有自己专职的变更团队。这样一个变更团队中的维护程序员数量,可以用以功能点计算的软件规模除以上文所述适当的维护任务量范围而得。)

缺陷修复率 一个自然月 22 个工作日内单个维护程序员能够修复的缺陷或错误平均数。美国平均为每个自然月修复 10 个缺陷,其变化范围从每人月修复少于 5 个缺陷到修复约 17 个缺陷不等。影响缺陷修复率的因素包括软件维护程序员的工作经验、代码复杂度及不良修复注入或者修复前一个缺陷时意外引入所创建代码的新缺陷。美国的平均不良修复注入率为 7%。

革新生产率 使用全套革新支持工具所进行的软件应用革新的每人月功能点平均值。美国平均为每人月 65 个功能点。其变化范围从用晦涩难懂的编程语言编写的高度复杂应用软件的每人月 25 个功能点到相当现代化的编程语言编写的中等复杂度应用软件的超过每人月 125 个功能点不等。影响革新生产率的其他因素包括应用软件的整体规模、应用软件中是否含有“易错模块”以及革新团队的工作经验等。

(如果手工革新没有自动化支持,执行起来则要困难得多,因此生产率也低得多——只有每人月 14 个功能点左右。虽然这比某些新应用开发还高一些,但就投资回报而言,仍然接近投资边际。)

应用软件不会优雅地老去。一旦被投入到生产环境,软件会以如下 3 种方式持续地变化:

1. 软件部署之后,发布的软件中仍然存在的潜伏缺陷一定会被发现和修复。
2. 要么因为商业需要,要么新的法律法规要求,或者两者都有,应用软件的规模会以每年 5% 到 10% 的速率持续增长并不断增加新功能。
3. 缺陷修复和功能增强的双重叠加,往往会逐渐降低应用软件的结构性,并增加其复杂度。这种描述软件复杂度随时间而增加的术语称为“熵”。软件熵值增加的平均比率为每

年约 1% ~ 3%。

某些应用软件使用多种编程语言编写，这导致了软件应用的一个特殊问题。曾经在一个单一应用中笔者发现了多达 15 种不同的编程语言。

多种编程语言使软件维护变得非常棘手，因为这增加了软件维护团队学习众多编程语言的繁重负担。同时，从某种意义上说，这些编程语言中的一些（或者全部）可能已经“死亡”，因而它们已经没有任何仍然可以工作的编译器或解释器了。这种情况严重抑制了软件维护的生产率，增加了出现不良修复注入的可能性。

因为软件缺陷去除和质量控制不够完善，在已交付的软件应用中总会有软件缺陷或错误需要修复。当前，美国平均的缺陷去除效率只有开发期间引入错误或缺陷的大概 85%。而软件行业保持这个平均水平已经超过 20 年了。

开发期间所写代码的实际潜在缺陷为每个功能点约 5 个缺陷。如果在发布之前发现并修复这些缺陷中的 85%，那么每个功能点仍有约 0.75 个缺陷会发布给客户。

对于规模在 1000 个功能点级别或 100 000 个源代码语句数量级的典型应用软件，这意味着在交付之后将仍然存在约 750 个缺陷。其中约四分之一，即 185 个缺陷，会严重到足以导致应用停止运行或产生错误输出。

既然潜在缺陷往往随着应用软件的整体规模的增加而增加，而缺陷去除效率水平随着应用规模大小增加而降低，随应用一起交付的潜伏缺陷整体数量也会随着应用规模的增加而增加。这就解释了为什么诸如 Microsoft Windows 和很多 ERP 应用软件等规模在 100 000 个功能点数量级的超大型应用软件，需要很多年时间才会达到相对稳定状态。这些大型应用软件都是带着成千上万的潜伏错误或缺陷交付给客户的。

当然，普遍情况要比最佳实践情况糟糕得多。正式审查、静态分析和自动化测试相结合，可以将累积缺陷去除效率水平提升到 99%。诸如团队软件过程（TSP）等方法可以将潜在缺陷降低到每个功能点 3 个缺陷以下。

除非遵从非常成熟的软件开发实践，否则，在一个新应用软件发布之后的第一年，软件团队的精力将高度集中于缺陷修复工作而只有极少数功能增强工作。

不过，数年之后，随着绝大多数初始缺陷都被发现和清除，应用软件可能已变得非常稳定。与此同时，新功能数量将会逐渐增多。

这种变化趋势的直接结果是，软件维护活动由初始的高度集中于缺陷修复工作逐渐转移到长期集中精力于增加新功能或增强现有功能上。

不仅软件带着大量的潜伏缺陷被部署出去这一现象，而且不良修复注入这一现象也已被观察了超过 50 年了。当前，所有缺陷修复的大约 7% 会引入一个以前不存在的新缺陷。对于非常复杂而结构化又较差的应用软件，不良修复注入的概率可高达 20%。

更令人担忧的是，不良修复一旦发生，要想纠正这种错误是非常困难的。虽然美国的平均初始不良修复注入率只有 7% 左右，但针对初始缺陷修复的不良修复造成的二次不良注入率达到 15%，而三次不良修复注入则高达 30%。笔者曾经观察到一连串的 5 次连续不良修复，每次修复尝试都引入了新的问题却没能成功修复初始问题。最后，第六次修复努力终于成功解决了初始问题。

20 世纪 70 年代, IBM 公司曾针对其主要商业应用软件做了一个客户所报告缺陷的分布分析。包括笔者本人在内, 参与这项研究的 IBM 人员惊讶地发现, 软件缺陷并不是平均地分布在大型应用的所有模块中的。

在 IBM 主要操作系统的案例中, 大约 5% 的模块包含了所有已报告缺陷总数 50% 以上的缺陷。最极端的例子是一个大型数据库应用, 245 个模块中的 31 个模块包含了所有客户报告缺陷中 60% 以上的缺陷。这些令人讨厌的地方就是著名的“易错模块”(error-prone module)。

AT&T 和 ITT 等其他公司所做的类似研究发现, 易错模块在软件领域普遍存在。20 世纪 80 年代和 90 年代早期, 在规模大于 5000 个功能点的应用软件中, 90% 以上的应用软件都发现含有易错模块。从众多公司获取的易错模块数据总结发表在笔者的书籍《Software Quality: Analysis and Guidelines for Success》里。

幸运的是, 一旦识别出易错模块, 就有可能通过外科手术去除它们。另外, 在软件项目中也可以采取措施防止这些易错模块出现。在这方面, 缺陷度量、正式设计审查、正式代码审查、正式测试和测试覆盖率分析均已被证明是行之有效的方法。

2009 年的今天, 易错模块在那些 SEI CMM 高于 3 级的组织里几乎已不存在。诸如团队软件过程(TSP)和 Rational 统一过程(RUP)等其他开发方法在预防易错模块方面也同样有效。而某些形式的敏捷开发方法(如极限编程 XP)在预防易错模块出现上也似乎有效。

易错模块去除是正常遗留应用改造工作的一个方面, 所以那些经过改造的应用程序在改造完成之后就不再含有易错模块了。

但是, 易错模块在 CMMI 等级 1 的组织中仍然比较常见和棘手。在那些没有经过革新及虽经维护却没有认真严格缺陷度量措施的遗留应用软件中, 易错模块仍然普遍存在, 这实在令人担忧。

一旦部署之后, 大多数应用软件的规模每年都会以其初始功能规模 5% 到 10% 的比率持续增长。某些应用软件, 如 Microsoft Windows, 历经十余年之后, 其规模可能会增加到原来的好几倍。

持续增加的新功能, 再加上连续不断的缺陷修复, 通常都会使本已老态龙钟的应用软件的复杂度水平不断升高。结构复杂度可以通过控制流图循环数及许多商业化工具使用的基础复杂性等度量指标来加以度量。如果复杂度是以年度为单位进行度量的, 且没有蓄意保持低复杂度, 那么复杂度的增加率每年在 1% ~ 3% 之间。

然而重要的是, 软件熵或者复杂度增长的速率直接与该应用软件的初始复杂度成正比。例如, 如果一个应用软件发布时的平均控制循环复杂度等级低于 10, 那么该应用软件将会在历经至少 5 年的正常维护和功能增强变更之后仍然保持良好的软件结构。

但是, 如果一个应用软件发布时其平均控制循环复杂度水平大于 20, 则其软件结构将会快速恶化, 且其复杂度水平的增加可能每年超过 2%。经过几年之后, 其熵值变化率及复杂度甚至会加速增加。

碰巧的是, 不良修复注入和易错模块往往也都与较高的复杂度水平紧密相关(尽管不是完全相关)。大部分易错模块都具有 10 或者更高的控制循环复杂度水平。修改高复杂度应

用软件工作的不良修复注入水平经常高于 20%。

这里,软件革新同样可以降低软件熵值,将控制循环复杂度水平降低到 10 以下,而这正是代码复杂度的最大安全水平。

要精确探讨软件维护成本有诸多困难。其中困难之一是,软件维护任务经常被分配给软件开发人员。当需要时,开发和维护任务经常相互交叉进行。这种实践做法使区别软件维护成本和软件开发成本变得很困难,因为程序员经常在记录时间如何分配上相当粗心。

另外一个非常突出的问题是,大量软件维护任务只包含对软件应用进行很小的改动。相当一部分缺陷修复可能涉及只有一行的代码改动。添加一个微小的新功能,比如在屏幕上添加一行新内容,可能只需要不多于 50 个源代码语句。

这些小变化太小,还达不到正规功能点计算的最低有效度量阈值。功能点度量指标包括复杂度加权因子,即使将复杂度调整设置为功能点计算范围的下限,仍然难以计算 15 个功能点以下的这些微小代码变化。

为度量微小的软件维护变更和缺陷修复,人们开发出了被称为“微型功能点”的实验方法。该方法类似于标准的功能点计算方法,但精确度增加了 3 个数量级,因此可以计算单个功能点很小的一部分。

当然,完成一项由“微型功能点”度量的少量更改工作可能只需一小时或更少时间。但在那些每年要完成多达 20 000 个这样更改的大公司里,其累计成本可不容小觑。“微型功能点”可以用来解决那些微量维护更新无法进行正式经济分析的问题。

相当一部分软件维护任务所涉及的程序更改,要么只是一个功能点的一部分,要么至多不到 5 个功能点或大约 250 个 Java 源代码语句。尽管正常的功能点计算方法对于微小的软件更新并不可行,而微型功能点也仍然还是一种实验方法,但使用逆火(backfiring)方法将逻辑源代码语句数量转换为等效的功能点数目还非常可行。例如,假设一个小的软件更新需要向一个现有应用软件添加 100 个 Java 语句,由于通常将 50 个 Java 语句记为一个功能点,那么这就可以认为,这个小软件维护项目的规模就是两个功能点。

由于 23 种独立的软件维护工作经常相互交错,且大型和小型的更新维护工作经常相互混合在一起,软件维护工作比传统的软件开发工作更加难以评估和度量。结果,软件维护的基准研究也比软件开发的基准研究少得多。实际上,关于软件维护的可靠信息大大少于软件的几乎任何其他方面的信息。

软件维护工作经常被外包给美国国内或者国外的离岸外包公司。由于各种商业原因,当前的软件维护外包合同似乎更加稳定,与软件开发合同相比,更不太可能卷入官司。

软件维护外包合同的成功是由于以下两个主要因素:

1. 小的软件维护变更没有大多数软件开发项目的巨大成本和进度延误率。
2. 对现有软件的微小维护变更几乎从来没有完全失败过。而大量的软件开发项目则确实失败了,甚至有些项目从来就没有最终完成。

可能还有其他原因,但事实仍然是,软件维护的外包合同似乎更加稳定,相比软件开发外包合同,其不太可能卷入官司纠纷。

在 2009 年,软件维护是软件行业占主导地位的工作。在不确定的未来可能仍然会占据

主导地位。对于软件行业来说,与许多其他行业一样,一旦经历了50多年的发展之后,从事修复现存产品工作的人员数量就会比构建新产品的工人数量要多很多。

人口数据 在软件世界里,独立软件维护组织最常见于雇用了总数大概在500~50 000名软件人员的大型公司里。

据笔者估计,美国大约有2500个具有独立软件维护组织的大型公司。截至2009年,在美国,软件维护组织里从事着软件维护工作的软件人员数量可能有800 000人。(既从事软件开发工作又从事软件维护工作的软件人员数量则可能有400 000人。)

项目规模 软件缺陷的平均规模小于一个功能点,这也是需要微型功能点的原因。软件功能增强或新功能的典型规模范围为5个到500个功能点。然而,只要软件仍然还在使用,就会有众多的功能增强工作。这样,应用软件的规模就会以平均每年8%左右的速率增长。

生产率 由于难以找到确切的问题根源,再加上需要进行回归测试及构建新版本,缺陷修复的生产率只有大约每人月10个功能点。另一种表示缺陷修复效率的方法是使用每个月修复的缺陷或错误数,其平均值为每人月10个缺陷。

功能增强的生产率平均为每人月约15个功能点,但由于功能增强的性质和规模、团队的经验、代码的复杂度以及功能增强开发期间的需求变更速率等,不同的功能增强项目生产率也大相径庭。功能增强的生产率范围,可以低至每人月只有5个功能点,高达每人月35个功能点。

进度 缺陷修复的开发周期从几个小时到几天不等,只有一种情况例外。那些称之为“待定缺陷”或者修复团队无法重现的缺陷,可能需要花费数周时间才能予以修复,因为变更团队使用的应用内部版本可能不包含该缺陷。为了隔离出这些被暂时搁置的缺陷,有必要从用户那里获取更多信息。

修复一个缺陷与发布一个新版本并不一样。在IBM等一些公司里,在缺陷修复上,维护周期随所报告缺陷的严重性等级不同而不同;也就是说,严重性级别为1的缺陷(最严重),修复时间限定在大约1星期之内;严重性级别为2的缺陷,大约两星期;严重性级别为3的缺陷,在下一个发布版本中修复;严重性级别为4的缺陷,在下一个发布版本或者任何方便的时候予以修复。

功能增强的开发周期通常从1个月到9个月不等。然而,很多公司都有固定的发布间隔时间,这样,可以聚集很多功能增强和缺陷修复变更,然后在下一个发布里同时发布这些功能增强和缺陷修复。微软的“服务包”(Service Pack)就是个典型例子,同样的例子还有Firefox的间歇性发布版本。通常情况下,固定的发布时间间隔是每半年或一年一次,有些公司是每季度一次。

质量 关于软件维护或缺陷修复质量主要关注3个方面:(1)软件维护或功能增强的潜在缺陷比新开发软件的更高;(2)应用中是否存在易错模块,以及(3)缺陷修复的不良注入率平均约为7%。

软件维护或功能增强的潜在缺陷比新开发软件的潜在缺陷要高一些,达到每个功能点6.0个缺陷左右。缺陷去除效率通常比新开发软件低一些,只有约83%。结果是,交付缺陷达到平均每个功能点1.08个缺陷。

另一个数年内仍然会缓慢恶化的质量担忧是应用的复杂度（以控制流图循环复杂度度量）缓慢增加，因为应用软件的维护变更往往会使得初始的良好结构变得不再良好。结果是，每年，潜在缺陷都会比上一年略微升高，而不良修复注入也会增加。除非对应用软件进行彻底改造，否则这些问题往往变得极其糟糕，最终将不再能够安全地修改该软件。

除了改造，诸如用于重要功能增强和重大缺陷修复的正式审查、静态分析及自动化测试等方法均可将缺陷去除效率水平提升到 95% 以上。但是，不良修复注入和易错模块的存在仍然是个相当棘手的问题。

专业化 软件维护组织结构的主要目的就是支持软件维护的专业化。虽然并不是每个人都喜欢软件维护工作，但确实有相当多的程序员和软件工程师非常享受软件维护工作带来的乐趣。

软件维护组织中另一个专家工作包括集成和配置控制。尽管正式软件测试机构也会承担一些专门的测试工作，比如某个主要版本发布之前的系统测试，但维护软件工程师通常需要负责小型软件更新和微小功能增强的绝大多数测试工作。

令人奇怪的是，软件质量保证（SQA）专家很少参与由软件维护团队所执行的缺陷修复和微小功能增强活动。相比之下，SQA 专家经常需要在较大功能增强中承担责任。

技术文档作家在软件维护工作中也没有什么重要作用，但如果所做的功能增强同时需要对用户手册或帮助文档进行相应的变更，可能偶尔也需要技术文档作家参与此类项目。

即便如此，迄今为止，很少有真正涉及成功的软件维护程序员和成功的软件开发程序员之间个性和技术差异的研究。

警告与注意事项 关于软件维护专业和维护组织的主要警告是，它们往往会把从业人员锁定在一个狭窄的职业范围内，有时数年间只局限于维护某一款软件。如果软件工程师花费数年时间却只负责修复单一某一款应用软件的缺陷，那么这些人将几乎没有任何职业发展和知识扩展的机会。偶尔，让软件工程师在软件维护工作和软件开发工作之间来回切换，也不失为最大限度减少软件工程师职业倦怠的一个良好实践。

结论 与软件开发相比，有关软件维护组织的文献非常稀缺。尽管有一些很不错的书籍，但有关软件应用规模增加、熵值增长和缺陷趋势等历经数年的长期研究仍然不多。

大量研究和调查表明，截至 2009 年，软件维护依然是软件行业占据主导地位的软件活动。但在下列几个方面，仍然需要更多的研究：对遗留应用软件进行数据挖掘以提取业务规则；从遗留应用软件代码中去除安全漏洞；软件革新的成本和价值；在遗留代码上诸如审查、静态分析及自动化测试等质量控制方法的应用。

5.7.2 客户支持组织

在那些仅拥有少量应用软件及客户或用户的小型公司里，客户支持工作可能非正式地由开发团队自己负责。但是，随着客户数量的增加和需要支持的应用软件数量的增长，很快就需要正式的客户支持团队。

客户支持的非正式经验法则表明，客户支持团队的人员编制取决于以下 3 个变量：

1. 客户数量

2. 已发布软件中潜伏缺陷或错误数量
3. 以功能点或代码行数度量量的应用软件规模

满足下述标准的应用软件，很可能需要一个全职的客户支持人员：150 个客户，软件中有 500 个潜伏缺陷（75 个严重缺陷），以及 10 000 个功能点或者 Java 等编程语言的 500 000 条源代码语句。

已知的改善客户支持的最有效方法是实现比当今软件质量典型水平好得多的应用软件质量水平。软件交付时每减少大约 220 个潜伏缺陷，就可以减少 1 名所需的客户支持人员。这个结论基于这样的假设：每个客户支持人员每天需要跟大约 30 个客户交谈，每个发布缺陷会被 30 个客户遇到。结果，每 1 个已发布缺陷将占用 1 名客户支持人员 1 天，每年就是 220 个工作日。

某些公司试图通过对客户的支持电话呼叫收取费用甚至客户报告了缺陷也要对客户收费这种方式来降低客户支持成本！这是一个极其糟糕的商业实践，它极大地冒犯了客户，却对公司没有任何好处。每一个遭遇了因需要客服支持而被收取费用的客户都不会满意，这会让他们更加主动地去寻求更加合理、更具竞争力的产品。^①

同时，由于软件通常都是带着成百上千的严重缺陷交付出去的，且客户所报告的这些缺陷对软件厂商颇具价值，收取客户支持费用基本上切断了一个降低软件维护成本的有价值渠道。几乎没有哪个收取客户支持费用的公司拥有大量开心的客户，很多这样的公司也正在逐渐失去市场份额。

遗憾的是，客户支持组织是任何各种软件组织中人员编制和组织管理最困难的一个。造成这种结果有诸多原因。首先，除非公司向客户的支持申请收取费用（并不推荐），否则客户支持团队的成本会很高。其次，客户支持工作往往只有极其有限的职业发展机会，这使得客户支持工作很难吸引和留住人才。

因此，客户支持往往是最先外包给低成本海外离岸供应商的商业活动之一。又因为客户支持也是劳动力密集型工作，它也成了最先试图对至少某些电话呼叫应答进行自动化的商业活动之一。为了尽量减少客户与在线支持人员讨论所需的时间，还提供了各种各样的常见问题解答（Frequently Asked Questions, FAQ）以及各种可以通过电话或电子邮件访问到的帮助主题，以供客户在接通客服电话之前自助解决问题。

遗憾的是，这些自动化技术经常令用户感到沮丧不已，因为在接通一个在线真人客服的电话应答之前，用户往往需要花费数分钟来处理晦涩难懂的语音消息。更糟糕的是，这些自动化的语音消息对听力有障碍的人来说几乎毫无用处。

即便如此，从事客户支持业务的公司已经在语音应答系统上进行了很多有趣的技术创新，同时还开发出了很多相当成熟的用户技术支持（help-desk）在线帮助软件包，这些软件包可以持续跟踪用户电话呼叫或者电子邮件，识别以前已经报告过的错误或缺陷，以及其他辅助管理功能。

由于来自客户的电话或电子邮件包含了很多关于深层次缺陷或安全漏洞的潜在有价值

① 言外之意是，客户会转向其他竞争对手的产品从而造成客户流失。——译者注

信息，有远见的公司乐于捕获这些信息并作为其质量和安全改进计划的一部分而加以分析和使用。

一个称为“服务与支持专业协会”（SSPA）的社会组织，不仅为客户支持人员提供有用信息，也评估各种公司的客户支持业务，并为优秀者颁发各种奖项和表彰。SSPA 组织还举办各种关于客户支持的会议和活动。（SSPA 的官方网站为 <http://www.thesspa.com>。）

SSPA 曾与著名的 J.D. Power and Associates 公司^①合作，调研评估各个公司的客户服务，并颁发各种优秀奖项以激励各个公司提供更好的客户服务。例如，SSPA 网站上展示了下列截至 2009 年的某些奖项：

- ProQuest Business Solutions —— 最大进步奖
- IBM Rochester —— 连续 3 年持续卓越奖
- Oracle 公司 —— 创新支持奖
- Dell —— 关键任务支持奖
- RSA 信息安全公司 —— 复杂系统最佳支持奖

相对于销售软件的商业公司，信息技术基础架构库（ITIL）的大量资料汇编阐述了公司内部客户支持的一些重大主题，比如客户技术支持（Help-Desk）响应时间目标、服务协议、事件管理及数以百计的其他信息条目。

软件客户支持工作是由一种多级安排来组织管理的，初始级别使用自动化和常见问题解答（FAQ），其他级别则需要使用更多的专业知识。这种多层次安排类似于以下例子：

- 级别 0 —— 自动语音信息、FAQ 及可获取的帮助信息引导。
- 级别 1 —— 了解应用软件基础知识和常见缺陷的人员。
- 级别 2 —— 特定领域专家。
- 级别 3 —— 开发人员或最顶尖专家。

这种多层次客服支持方法背后的基本思想是最大限度地减少所需开发人员和专家的客户支持时间，同时希望以有效方式为客户提供尽可能多的有用信息。

正如本书很多地方提到过的，绝大多数客户服务电话和电子邮件都是由于软件糟糕的质量和过多的缺陷引起的。因此，使用诸如团队软件过程（TSP）、正式审查、静态分析、自动化测试等更加成熟的开发方法不仅可以降低开发成本、缩短开发进度，而且可以减少软件维护和客户支持成本。

有趣的事情是，让我们看看 J.D.Power 奖的获得者之一，IBM Rochester 是如何进行客户支持的：

“就客户服务响应时间和提供解决问题的能力而言，IBM 强烈关注客户支持的响应情况。当客户的电话打进来时，必须在一定时间内（一分钟或者几分钟）进行应答。IBM 不希望客户拿着电话花费超过 10 分钟以上时间，只是为了等着客服接电话。

当报告了问题、缺陷时，正式修复可能需要花费一些时间。在正式修复可用之前，客

① J.D. Power and Associates 公司是一家全球性的市场调查咨询公司，主要就客户满意度、产品质量和消费者行为等方面进行独立公正的调研。其 2005 年 4 月 1 日加盟 McGraw-Hill 公司，成为 McGraw-Hill 旗下的一个独立品牌。——译者注

服团队将尽快提供临时解决方案，并使用一个称之为“问题初次修复时间”的关键度量指标。对于一些新问题，初次临时修复可能花费不超过24小时，而对于某些已知问题，所需时间甚至更少。

当提供了正式修复时，IBM Rochester使用的另一个关键度量指标是缺陷修复质量：有缺陷的修复的比率。在IBM的各个主要平台上，IBM Rochester的有缺陷修复率是最低的。（虽然软件行业平均不良修复注入率只有7%，但IBM彻底解决了这个问题，这仍然值得称道！）

IBM Rochester支持中心还进行一项“跟踪调查”。这是一个关于IBM所提供服务或问题修复的客户满意度调查。这些调查基于那些已关闭的问题记录抽样结果。在这些跟踪调查中，IBM Rochester的满意客户比率占到90%以上。

另一个IBM Rochester获得奖项的原因之一是“文化因素”。IBM作为一家大公司及Rochester作为其一个实验室，均有长期关注产品质量的传统（比如，赢得了美国波多里奇国家质量奖）。由于客户满意度与产品质量直接相关，IBM Rochester实验室的产品长期拥有质量卓越的美誉（IBM System/34、System/36、System/38、AS/400、System i，等等）。IBM及Rochester的雇员对他们所交付产品和服务的卓越品质感到无比自豪。”

对于重大的客户问题，各个团队（支持、开发、测试等团队）将并肩工作以提出解决方案。IBM Rochester实验室长期从客户反馈中收益颇多，这也解释了IBM Rochester实验室连续多年荣获卓越客户支持奖的原因。当对客户进行调查时，客户明确而赞许地提到了他们所接受的来自IBM Rochester实验室的客户支持和解决问题的数量。

人口数据 在软件世界里，由于2008年的经济衰退，由自己员工组成的内部客户支持的数量正在快速下降。可能仍有几百个大型公司提供这样的客户支持服务，但随着公司裁员和机构精简的不断升级，其数量将会大大减少。

对于那些从未雇用任何专职客户支持人员的小型公司，毫无疑问，软件工程师们仍将继续接听客户电话并回复客户的电子邮件。大概10 000家或更多的雇员人数在1~50名之间的美国公司或组织，其客户支持任务是由软件工程师或程序员非正式地负责的。

对于商业软件组织，将客户支持工作外包给专业的客户支持公司已司空见惯。这些客户支持公司既有美国国内的公司，也有其他国家的几十家客户支持组织，那里的人力成本比美国或欧洲都要低。但是，随着经济衰退的持续，美国的人力成本也将下降，而这提供了大量未就业的软件技术人员储备。客户支持、软件维护和其他劳动力密集型工作机会可能将开始回流美国。

项目规模 那些具有强制性正式客户支持的应用软件，其平均规模为10 000个功能点。当然，对于任何规模的应用软件，客户都会有这样那样的问题且需要报告应用软件中的缺陷。但是，规模在10 000个功能点数量级的应用软件常常拥有大量客户。此外，这些大型系统也总是带着成千上万的潜伏缺陷发布出去。

生产率 客户支持工作的生产率不是以功能点而是以获得帮助的客户数量来度量的。典型地，电话支持的一级客户支持人员每天可以接听约30个客户的电话，换算成每个客户电话，约可交谈16分钟。

对于使用了大量专家的二级或三级客户支持,与客户交谈的工作可能不是全职的。然而,对于那些严重到需要二级客服支持的严重问题,专家的每次电话交谈需要花费 70 分钟。对于那些严重到需要三级客服支持的问题,毫无疑问,需要来来回回的多次电话沟通,可能还有一些内部研究要做。三级专家的电话沟通需要大约 240 分钟。

如果一个客户报告了一个以前从来没有识别出来或者修复过的新缺陷,那么可能需要数天甚至数周的时间予以修复。(笔者曾在一个诉讼案件中担任专家证人,在这个案件中,修复一个财务软件的一个缺陷所需要的时间超过了 9 个月。在修复此缺陷的过程中,前 4 次努力每次花费 2 个月。他们不但没能修复最初的缺陷,反而每次修复尝试都引入了新的缺陷。)

进度 客户支持的主要进度问题是在接通在线支持人员之前客户的电话等待或保持时间。2009 年的今天,客户要想接通在线客服人员需要花费从 10 分钟到超过 60 分钟之间不等的保持时间。不用说,这对客户来说是非常令人沮丧的。改进软件质量也可以减少客户的电话等待时间。假定客服人员数量不变,软件中每减少 10 个严重性等级为 1 或者 2 的发布缺陷,客户的电话等待时间就将减少约 30 秒。

质量 客户支持电话呼叫数量与软件中发布缺陷或错误的数量直接相关。发布零缺陷软件也会将客户支持电话呼叫数量减少为零,这在理论上是有可能的。当今世界,缺陷去除效率只有平均 85% 左右,而当软件发布出去的时候仍然存在成千上万的严重缺陷,因而,仍然将会有数以百计的客户支持电话呼叫和电子邮件。

有趣的是,某些开源或免费软件,比如 Linux、Firefox 及 Avira^①,似乎比诸如微软、Symantec 等广受认可的软件厂商发布的同类应用具有更好的质量水平。之所以如此,部分原因是开发者的技术技能优秀,部分原因是日常使用的工具优秀,比如发布之前的静态分析等。

专业化 一线客户支持的角色是非常专业化的。当同暴躁的客户打交道时,有效的客户支持服务人员需要有一个好的个性,再加上相当成熟的技术技能。这两者之中,技术标准要比当与生气或怒气冲冲的客户打交道时良好的个性标准更容易满足。话虽如此,迄今为止,涉及客户支持组织中客服人员个性及技术技能主题的相关研究仍然很少。

除了软件厂商提供的客户支持,一些用户协会和非营利性组织也会基于自愿提供某些客户支持服务。很多自由软件和开源应用都有众多能够回答相关技术问题的用户组。即使商业软件,有时获得来自某个专家级用户对一个问题的非正式响应也比来自该软件开发厂商的响应容易得多。

警告和注意事项 关于客户支持工作的主要警告是,它往往把相关人员锁定在很狭窄的职业范围内,有时局限于比如 Oracle 或 SAP 等某个单一软件长达数年之久。而这使得客服人员很少有职业成长的机会或知识扩展的空间。

另一个警告是,通过自动化和专家系统以改善客户支持在技术上是可行的,但很多现

① Avira 是一款著名的德国杀毒软件,中文昵称“小红伞”,其英文名为 AntiVir。详细信息请参见 Avira 官方网站 <http://www.avira.com/zh-cn/for-home> ——译者注。

存的专利已经涵盖了这些主题。结果是，开发更好的客户支持自动化系统的尝试可能需要得到相应知识产权的许可。

结论 客户支持文献主要包括两种迥然不同的信息。信息技术基础架构库（ITIL）包括超过 30 本书和超过 5000 页的信息，涵盖客户支持的每一个方面。但是，ITIL 库主要针对公司内部的支持，它们讲述的并未被商业软件供应商大量采用。

对于商业软件的客户支持，有很多行业书籍可用，但这些文献往往主要是客户支持外包公司所发布的白皮书或专题论文。尽管这些白皮书或专题论文往往都是营销说辞，但是其中的某些文章确实可以提供关于客户支持机制的有用信息。还有很多来自那些提供客户支持自动化公司的有趣报告，这些报告既内容丰富，又涉及众多方面的功能。

大量研究和调查表明，2009 年，客户支持工作是软件行业重要的软件活动。在下列几个方面，目前仍然需要大量的研究：软件质量和客户支持之间的关系、用户协会和志愿者团体的角色作用以及可能会改善客户支持工作的潜在自动化方法。尤其是，在为耳聋或有听力障碍的客户、失明客户以及有其他身体障碍的客户提供支持服务方面，仍需更多的研究。

5.7.3 软件测试组织

在讨论软件测试组织时，有 10 个问题需要特别指出：

1. 现存有超过 15 种不同的软件测试类型。
2. 基于公司的测试策略，很多类型的测试工作均可以由开发者自己、内部测试组织、外包测试组织或者质量保证团队来执行。
3. 在敏捷团队和传统层级式组织结构下，测试员可能与开发者共同属于同一团队而没有自己独立的部门。
4. 在矩阵式组织结构下，测试员可能属于一个独立的测试部门，他向技能经理汇报，但根据需要分配给一个特定的项目使用。
5. 一些测试组织是质量保证组织的一部分，因此，除了包含测试专家之外，还有一些其他种类的专家。
6. 一些质量保证组织收集软件测试的数据结果，但不亲自做测试工作。
7. 一些测试组织被称为“质量保证”部门，但只做软件测试工作。这些测试组织并不从事其他质量保证活动，比如审查管理、质量度量、质量预测、质量培训等。
8. 对于任何给定的软件应用，基于公司的测试策略，其独立测试种类数目，从 1 种到 17 种测试类型不等。
9. 对于任何给定的软件应用，基于公司的质量策略，其测试组织和（或）作为其测试策略一部分的质量保证组织的数量，从 1 个到 5 个不等。
10. 对于任何具体的缺陷去除活动，包括测试在内，可能需要多达 11 种不同专家参与其中。

从上述特别强调的 10 点可以推测出，2009 年，软件行业不存在应用软件测试的标准方法。不仅没有标准的测试方法，而且没有测试覆盖率或缺陷去除效率的标准度量方法，尽

管这两者都是简单易懂的度量。

最广泛使用的测试度量方式是测试覆盖率，它表示测试用例所实际执行的代码数量。测试覆盖率度量已完全自动化，因而执行起来并不难。这个度量指标非常有用，但与此同时，度量缺陷去除效率对软件项目来说则更有价值。

缺陷去除效率的度量更为复杂，且其并未完全自动化。为度量特定测试阶段（比如单元测试）的缺陷去除效率，需要记录测试发现的所有缺陷。单元测试完成后，所有其他测试阶段发现的缺陷也要记录，包括产品发布后前 90 天内客户报告的缺陷。所有缺陷一合计，就可以很容易地算出各个阶段的缺陷去除效率。

假设单元测试发现了 100 个缺陷，功能测试和随后的测试阶段发现了 200 个缺陷，客户在产品发布后前 90 天内报告了 100 个缺陷。那么，发现的缺陷总数为 400 个。既然单元测试发现了这 400 个缺陷中的 100 个，在这个例子中，它的缺陷去除效率就是 25%，实际上，这离单元测试缺陷去除效率的平均值 30% 并不太远。

（一个快速确定缺陷去除效率但不太可靠的方法是缺陷植入。例如，如果在上述讨论的应用软件中预先植入 100 个已知缺陷，而最终发现了其中的 25 个，那么立刻就能算出缺陷去除效率为 25%。但是，无法保证能够以精确相同的比率发现这些预先植入的“已驯服”缺陷和那些意外引入的“野生”缺陷。）

遗憾的事实是，大多数形式的测试都并不是非常有效，它们只能发现实际存在缺陷的约 25% ~ 40%，虽然其实际范围从不到 20% 到超过 70% 不等。

有趣的是，软件行业对黑盒测试、白盒测试和灰盒测试仍有诸多争论。黑盒测试缺乏软件的内部信息，白盒测试则使程序内部代码完全可见，而灰盒测试具有内部代码结构的可见性但只在外部层面进行测试。

到目前为止，可以确定的是，这些争论仍是理论性的，并且有人已经进行了数个实验来度量黑盒测试、白盒测试和灰盒测试的缺陷去除效率水平。从度量实验结果可知，白盒测试似乎比黑盒测试具有更高水平的缺陷去除效率。

由于许多单独测试阶段（比如单元测试）的效率如此之低，这就可以看出为什么需要几种不同类型的测试并驾齐驱。术语“累积缺陷去除效率”指所有测试或缺陷去除活动的整体效率。

由于缺乏软件测试标准，缺乏广泛使用的测试有效性度量方法，对于实现高水平软件质量，软件测试本身似乎并不是个特别具有成本效益的方法。那些纯粹依赖于软件测试以去除缺陷的公司，其累积缺陷去除效率几乎从来没有高于过 90%，常常是低于 75%。

最新的测试方法，比如测试驱动开发（Test-Driven Development, TDD），使用测试用例作为规格说明的一种形式，在实际编写程序代码之前优先创建测试用例。结果，TDD 的缺陷去除效率比很多其他测试形式都要高，高达 85%。但是，即使是 TDD 方法，仍需考虑不良修复注入因素。大约 7% 的缺陷修复尝试会意外地在修复代码里面包含新的缺陷。

如果 TDD 与其他方法相结合，比如测试用例的正式审查和产品代码的静态分析，则整体缺陷去除效率可高达 95%。

关于自动化测试和手工测试的对比数据饱含歧义。理论上，自动化测试应该至少在

70% 的实验中比手工测试具有更高的缺陷去除效率。例如,手工单元测试平均只有 30% 的缺陷去除效率,而自动化单元测试则高达 50%。但是,该课题的更多研究表明,软件工程师和程序员之间的测试技能千差万别,因而自动化测试的效率也差别巨大。

常规测试的缺陷去除效率并不高这一事实为我们提出了一个重要问题:如果软件测试在发现和去除缺陷上并不是非常有效,那么真正有效的该是什么呢?这是一个非常重要的问题,也恰恰是在这本名为“软件工程最佳实践”的书中需要回答的一个问题。

“在实现高水平软件质量上什么是最有效的”之类问题的答案是:为达到最佳缺陷去除效率,需要组合使用缺陷预防和多种缺陷去除方法。

缺陷预防指能够将潜在缺陷从当前美国平均每功能点 5.0 个缺陷降得更低的各种方法和技巧。在缺陷预防上已有明确效果的方法包括软件能力成熟度模型集成(CMMI)的较高等级、联合应用设计(JAD)、质量功能展开(QFD)、根本原因分析方法、软件六西格玛方法、团队软件过程(TSP)以及个体软件过程(PSP)等。

对于小型应用软件,用户作为团队一员而参与开发团队工作的敏捷方法也可以有效降低潜在缺陷。(对于用户参与的情况,需要说明的是,对于一个拥有超过 50 名用户的应用软件,1 名用户并不能代表所有用户的实际情况。对于那些具有成千上万用户的应用软件,仅有单一用户参与项目是远远不够的。在这种情况下,焦点小组和众多用户的调查研究将非常必要。)

而需求、设计和代码的正式审查承担了双重职责,在缺陷预防和缺陷去除方面均非常有效。这是因为,正式审查的参与者能够自觉地避免犯在正式审查中发现的同样错误。

已被证明可以提高缺陷去除效率水平的方法组合包括:需求、设计、代码和测试材料的正式审查;测试之前的代码静态分析;包括至少 8 种测试类型的软件测试序列:(1)单元测试,(2)新功能测试,(3)回归测试,(4)性能测试,(5)安全测试,(6)可用性测试,(7)系统测试,(8)某些与客户或用户共同进行的外部测试形式,比如 Beta 测试或者验收测试。

就累积缺陷去除效率水平而言,预测试审查、静态分析及至少 8 种独立测试阶段的组合使用,通常可以将缺陷去除效率水平提高到 99%。这种组合不仅能够提高软件缺陷去除效率水平,而且还极具成本效益,相得益彰。

缺陷去除效率水平高达 95% 的项目通常比那些在质量方面投入吝啬的项目具有更短的开发周期和更低的发展成本。当然也具有低得多的软件维护和客户支持成本。

软件测试是一个可传授的技能,目前有很多营利性和非营利性组织为软件测试人员提供各种研讨会、课程以及各种特色的认证。尽管有证据表明,经过认证的测试人员最终比未经认证的测试人员具有更高的缺陷去除效率水平,但软件行业糟糕的度量基准研究实践使得上述主张有点儿像是奇闻轶事。如果测试认证包括了如何度量缺陷去除效率的部分内容,软件行业将受益匪浅。

表 5-3 所示为各种不同形式的软件审查、静态分析、测试及可能执行这些活动的组织或人员的示例。

表 5-3 软件缺陷去除活动的形式

预测测试审查		执行者	预测测试审查		执行者
1	需求	分析师	14	可用性测试	人类因素专家
2	设计	设计师	15	组件测试	测试员
3	编码	程序员	16	集成测试	测试员
4	测试计划	测试员	17	本地化测试	外国语言专家
5	测试用例	测试员	18	平台测试	平台专家
6	静态分析	程序员	19	SQA 验证测试	软件质量保证人员
常规测试			20	实验室测试 (Lab Testing)	硬件专家
7	子程序测试	程序员	外部测试		
8	单元测试	程序员	21	独立测试	外部测试公司
9	新功能测试	测试员或程序员	22	Beta 测试	客户
10	回归测试	测试员或程序员	23	验收测试	客户
11	系统测试	测试员或程序员	特殊活动		
特殊测试			24	审计	设计师、SQA
12	性能测试	性能专家	25	独立确认与验证 (Independent Verification & Validation, IV&V)	IV&V 承包商
13	安全测试	安全专家	26	道德黑客入侵 (Ethical hacking)	黑客入侵 (Hacking) 顾问

表 5-3 显示了 26 种不同种类的缺陷去除活动，由总共 11 种不同种类的内部专家、3 种外部公司的专家以及客户来执行。但是，只有非常大型且成熟的高科技公司才会拥有如此种类丰富的专家资源，使用如此众多不同种类的缺陷去除活动。

小型公司要么让其软件工程师或程序员（通常并未受过良好的测试培训）来执行测试工作，要么拥有一个主要由测试专家组成的测试团队。尽管截至 2009 年，这种情况并不多见，但软件测试工作确实也可能被外包出去。

这里，提出测试文献中并未很好地涵盖的 3 个主题将非常有益：

1. 各种测试类型分别需要多少测试员？
2. 各种测试类型分别需要多少测试用例？
3. 各种测试类型的缺陷去除效率分别是多少？

表 5-4 显示了表 5-3 中所示的 17 种测试类型的粗略人员编制水平。需要注意的是，这里的信息只是近似，实践中每一种测试类型的实际人数可能变化巨大。

由于软件测试需要执行源代码，因而表 5-4 中的信息基于源代码数而非功能点数。鉴于目前从汇编语言到现代编程语言（如 Ruby 或者 E 语言）共有超过 700 余种编程语言，表 5-4 所示的相同应用程序在源代码规模方面对于不同编程语言可能会有超过 500% 的差异。表 5-4 中使用的是 Java 语言，因为 Java 语言是 2009 年最常见的编程语言之一。

“任务量范围”列表明一个测试员可能需要负责测试的源代码数量。注意，根据测试人

员的经验水平、代码的控制流程图循环复杂度以及某种程度上软件程序中被测试的特定编程语言或语言组合的不同,任务量范围差异非常大。

由于表 5-4 中所示测试牵涉的众多不同技能的不同人员可能来自不同部门,因此将所有 17 种测试所需要的测试人员组成分解如下:5 名负责单元测试的开发者;两名集成测试和系统测试的测试专家;3 名负责安全、本地化和可用性测试的专家;1 名 SQA 专家;7 名来自其他公司的外部专家;以及两名客户。共计 20 人。

当然,任何 1000 个功能点或 50 KLOC (千行代码) 的小应用软件都不太可能使用(或需要)所有 17 种测试类型。对于一个 50 KLOC 的 Java 应用软件,最有可能的组合是 6 种测试类型,由 5 名开发者、2 名测试专家和两名用户共 9 名测试人员负责执行测试工作。

表 5-5 使用以上各表中的数据作为人员组成基础,但该表的目的是为了展示各个测试阶段所产生的测试用例近似数量,以及整个应用软件的测试用例总数。同样,其跟实际项目会有很大变化,所以该数据只是近似值。

Java 应用软件 50 KLOC 抽样代码的潜在缺陷为总共约 1500 个缺陷,等于每个功能点 1.5 个代码缺陷,或每千行代码含有 30 个缺陷。(注意:需求和设计中的早期缺陷被排除在外,且假定在测试之前已被去掉。)

如果使用所有 17 个测试类型,它们可能会检测到所有存在缺陷的 95%,即总共 1425 个缺陷。当应用软件交付给客户时,仍有 75 个潜在缺陷遗留在软件中。假设潜在缺陷数目和测试用例数目都相当精确(非常可疑的假设),那么平均需要 1.98 个测试用例就能发现 1 个缺陷。

当然,既然通常只使用了 17 个测试阶段的 6 个,其缺陷去除效率也只有接近 75%,这就是为什么需要额外的非测试方法,比如正式审查和静态分析,以真正实现较高的缺陷去除效率水平。

表 5-4 给定测试阶段的测试人员编制

	软件语言	Java	
	软件代码规模	50 000	
	软件千代码行 (KLOC)	50	
	软件功能点数	1 000	
	常规测试	任务量范围	测试人员编制
1	子程序测试	10 000	5.00
2	单元测试	10 000	5.00
3	新功能测试	25 000	2.00
4	回归测试	25 000	2.00
5	系统测试	50 000	1.00
	特殊测试		
6	性能测试	50 000	1.00
7	安全测试	50 000	1.00
8	可用性测试	25 000	2.00
9	组件测试	25 000	2.00
10	集成测试	50 000	1.00
11	本地化 (Nationalization) 测试	150 000	0.33
12	平台 (Platform) 测试	50 000	1.00
13	SQA 验证测试	75 000	0.67
14	实验室 (Lab) 测试	50 000	1.00
	外部测试		
15	独立测试	7 500	6.67
16	Beta 测试	25 000	2.00
17	验收测试	25 000	2.00

表 5-5 给定测试阶段的测试用例数

软件语言		Java		
软件代码规模		50 000		
软件千行代码 (KLOC)		50		
软件功能点数		1 000		
	测试人员	每 KLOC 测试用例	总测试用例	每人测试用例
常规测试				
1 子程序测试	5.00	12.00	600	120.00
2 单元测试	5.00	10.00	500	100.00
3 新功能测试	2.00	5.00	250	125.00
4 回归测试	2.00	4.00	200	100.00
5 系统测试	1.00	3.00	150	150.00
特殊测试				
6 性能测试	1.00	1.00	50	50.00
7 安全测试	1.00	3.00	150	150.00
8 可用性测试	2.00	3.00	150	75.00
9 组件测试	2.00	1.50	75	37.50
10 集成测试	1.00	1.50	75	75.00
11 本地化测试	0.33	0.50	25	75.76
12 平台 (Platform) 测试	1.00	2.00	100	100.00
13 SQA 验证测试	0.67	1.00	50	74.63
14 实验室 (Lab) 测试	1.00	1.00	50	50.00
外部测试				
15 独立测试	6.67	4.00	200	29.99
16 Beta 测试	2.00	2.00	100	50.00
17 验收测试	2.00	2.00	100	50.00
总测试用例数			2825	
每 KLOC 测试用例数			56.50	
每人测试用例 (20 测试员)			141.25	

如果这个 50 KLOC 的小例子使用了超过 2800 个测试用例, 很显然, 拥有数百个软件应用的公司最终将拥有数以百万计的测试用例。一旦创建出来, 这些测试用例就可在适当的时候用于回归测试。幸运的是, 软件行业现有大量自动化工具可用于存储和管理这些测试用例库。

存在如此巨大的测试库是软件开发和维护的必要成本。但是, 这方面需要更多额外的研究。创建可重用的测试用例似乎颇具价值。同样, 测试用例中也经常存在错误, 这也是为什么测试计划和测试用例的正式审查也同样如此有用。

由于大公司和政府机构里数以百计的不同人员创建了数量巨大的各种测试用例, 很有可能会偶然创建出重复的测试用例。实际上确实如此, IBM 的一项研究指出, 在一个软件实验室的测试库里, 存在大约 30% 的冗余或重复测试用例。

本部分内容的最后一个表 5-6 显示了 6 个软件错误来源的缺陷去除效率水平：需求缺陷、设计缺陷、编码缺陷、安全缺陷、测试用例缺陷和性能缺陷。

事实上，并非每一种缺陷去除方法对所有类型的缺陷都同等有效，因而表 5-6 显得有些复杂。实际上，许多缺陷去除方法对安全漏洞几乎完全无效。编码缺陷是最容易去除的缺陷类型；需求缺陷、安全缺陷和测试材料中的缺陷是最难以消除的缺陷类型。

表 5-6 各种缺陷类型的缺陷去除效率

预测试审查	需求缺陷	设计缺陷	编码缺陷	安全缺陷	测试缺陷	性能缺陷
1 需求	85.00%					
2 设计		85.00%		25.00%		
3 编码			85.00%	40.00%		15.00%
4 测试计划					85.00%	
5 测试用例					85.00%	
6 静态分析		30.00%	87.00%	25.00%		20.00%
常规测试						
7 子程序测试			35.00%			10.00%
8 单元测试			30.00%			10.00%
9 新功能测试		15.00%	35.00%			10.00%
10 回归测试			15.00%			
11 系统测试	10.00%	20.00%	25.00%	7.00%		25.00%
特殊测试						
12 性能测试	5.00%	10.00%				70.00%
13 安全测试				65.00%		
14 可用性测试	10.00%	10.00%				
15 组件测试		10.00%	25.00%			
16 集成测试		10.00%	30.00%			
17 本地化测试	3.00%					
18 平台测试		10.00%				
19 SQA 验证测试	5.00%	5.00%	15.00%			
20 实验室测试	10.00%	10.00%	10.00%			20.00%
外部测试						
21 独立测试		5.00%	30.00%	5.00%	5.00%	10.00%
22 Beta 测试	30.00%		25.00%	10.00%		15.00%
23 验收测试	30.00%		20.00%	5.00%		15.00%
特殊活动						
24 审计	15.00%	10.00%				
25 IV&V	10.00%	10.00%		10.00%		
26 道德黑客入侵				85.00%		

从历史上看，正式审查对最大范围的各种缺陷具有最高的缺陷去除效率水平。最新的静态分析方法对编码缺陷也有很高的缺陷去除效率水平，但当前该方法只在现存的 700 多

种编程语言中约 15 种语言上有所应用。

表 5-6 中的数据具有较高的边际误差，但该表本身给出了整个软件行业需要大量收集的那些数据种类，用以改进软件质量和提升整体缺陷去除效率水平。实际上，每一个规模大于 1000 个功能点的软件应用都应该收集此类数据。

表 5-6 中没有列出的一个重要缺陷来源是不良修复注入。大约 7% 的缺陷修复代码本身含有新的缺陷。假定单元测试发现和去除了一个应用程序的 100 个缺陷，由于缺陷修复代码本身的错误，非常可能会有 7 个新缺陷被意外注入到该应用中。（易错模块的不良修复注入率可能高达 25% 以上。）

不良修复注入是软件中非常常见的缺陷来源，但在软件测试文献和软件质量保证文献中均没有很好地涵盖到该主题。

另一个并未很好涉及的质量问题是易错模块。正如本书很多地方提到的，应用程序里的缺陷并不是随机分布的，它们往往会聚集在少量漏洞百出的模块里。

如果一个应用包含一个或多个易错模块，那么针对这些模块的缺陷去除效率水平可能只有表 5-6 中所示数值的一半，而不良修复注入率则可能高达 25%。这就是为什么通常很少会去修复易错模块，而多是通过外科手术式去除掉或者用一个全新模块彻底替换掉它。

尽管软件测试工作历史悠久，软件行业雇用了大量的软件测试从业人员，人们仍然需要更多关于软件测试的研究。某些需要深入研究的主题包括：从功能说明书到测试用例的自动生成、可重用测试用例开发、更好的测试用例数量和缺陷去除效率预测方法、就缺陷去除效率而言更好的测试结果度量方法。

人口数据 在软件世界里，长期以来，软件测试都是重要的软件开发活动之一，而且软件测试也是最大数量的软件职业团体之一。但是迄今为止，一直没有软件测试从业人员的准确人口普查数据，部分原因是太多不同种类的专家参与软件测试工作。

由于测试工作是软件发布的关键环节之一，当软件项目进度落后时，就会有一种倾向，那就是软件项目经理或甚至高级管理人员对测试人员施加压力以缩减测试工作。通过让测试组织汇报给独立的技能经理，而不是项目经理或者程序经理，可以一定程度上增加测试工作的独立性。

然而，软件测试是软件开发不可分割的一部分，软件测试人员需要在软件开发开始的第一天就完全地介入其中。测试员无论是汇报给技能经理还是隶属于项目团队，在需求和设计阶段他们都需要尽早地投入其中。对于测试驱动开发（TDD）尤其如此，因为测试用例是需求和设计过程不可或缺的一部分。

项目规模 具有强制性正式测试要求的应用软件的最小规模是 100 个功能点。一般来说，应用软件规模越大，就需要越多种类的预测试缺陷去除活动和越多种类的测试活动，以成功完成应用开发。

对于少于 10 000 个功能点的大型系统，为实现高水平的缺陷去除效率，需要使用审查、静态分析、安全分析以及 10 种测试类型等缺陷去除活动。不幸的是，许多公司在测试和非测试质量活动上投入相当抵门，所以美国的平均软件质量水平令人尴尬地糟糕：累积缺陷去除效率只有 85%。从 1996 年到 2009 年，这一结果相当平稳，几乎是一成不变。

生产率 对于软件测试，并没有有效的生产率，也没有有效的测试用例规模度量指标。从宏观层面看，测试生产率可以用“每个功能点的工作小时数”或者与其互补的“每人月功能点数”来度量，但这些度量方法都太过抽象，无法真正体现软件测试的本质。

诸如“每月创建的测试用例数”或者“每月执行的测试用例数”等度量方法更是传递了错误信息，它们可能鼓励更多的额外测试而使测试结果膨胀，但却没有提升任何缺陷去除效率。

诸如“每月发现的缺陷数”等度量方法也不可靠，因为对于真正顶尖的开发者，有可能无法在其开发的软件中找到很多缺陷。出于同样原因，指标“平均缺陷成本”也不可靠。无论应用软件中是否存在任何缺陷，测试员仍然需要运行很多测试用例。结果是，平均缺陷成本随着缺陷数量的减少而上升；因此，平均缺陷成本指标实际上对软件质量非常不利。

进度 测试人员的首要进度问题是测试用例的创建和执行。但测试用例相比，测试进度依赖于发现的缺陷数量和修复缺陷所花费的时间要更多一些。

一个很少被度量但却同样阻碍测试进度的因素是测试用例自身含有的错误或缺陷。多年以前 IBM 所做的一项研究发现，测试用例中的缺陷比这些测试用例要测试的应用软件中的缺陷还多。（这项研究和那个发现测试库中 30% 冗余或重复测试用例的研究是同一个研究。）但当前的测试文献并未很好地涵盖测试用例错误这一主题。运行重复的测试用例增加了测试成本、拖长了测试周期，但却不能提升缺陷去除效率水平。

当开始测试一个含有大量缺陷的应用软件时，项目的整体进度就充满了风险，因为测试进度将会远远超出项目规划的截止日期。实际上，由于过多缺陷而导致的测试进度落后常常是软件项目进度落后的主要原因。

尽可能保证测试进度的最有效方法是让应用软件中尽可能少地出现缺陷，因为测试开始之前的预测试审查和静态分析已经发现了绝大多数缺陷。诸如 TSP 或者联合应用设计 (JAD) 等缺陷预防方法也可以加快测试进度。

就软件行业整体而言，由于过多缺陷而导致的测试进度落后是软件应用成本超支、开发进度落后以及项目取消的主要原因。由于长时间的进度延误和项目取消已经引发了大量的诉讼案件，居高不下的潜在缺陷和较低的缺陷去除效率水平成了违反合同诉讼的主要诱发因素。

质量 在发现缺陷上，软件测试本身还不能足以高效地使其成为主流软件应用所使用的唯一缺陷去除方式。软件测试本身的缺陷去除效率水平几乎从来没有达到过 85%，而最新的测试驱动开发 (TDD) 方法除外，该方法的缺陷去除效率水平可达 90%。

相比单独使用软件测试这一种方式，组合使用软件测试、正式审查和静态分析，能够实现更高水平的缺陷去除效率、更短的开发周期、更低的项目成本。此外，这些节省不仅有益于开发工作，而且可以降低下游客户支持和软件维护的成本。

建议管理人员或者有资格签署软件合同的读者将 95% 作为可接受的最低缺陷去除效率水平。每一个外包合同、每一份内部质量计划以及与软件供应商的每一项许可协议，都应该要求提供证据以证明开发组织在缺陷去除效率方面能够高于 95%。

专业化 测试专业涵盖了广泛的技能。然而，对于很多奉行“通才哲学”的小公司，

其开发人员可能也承担着软件测试员的角色,即使他们可能没有受到过关于软件测试角色的正确、良好培训。

对于大型公司,由测试专家组成的正式测试部门将比开发者自己做测试带来更好的测试结果。对于超大型跨国公司和那些构建系统软件和嵌入式软件的公司,软件测试和质量保证专家为数众多,而且每个专家都具有多种多样的技能。

当前有多种形式的软件测试认证可用。那些为了很好地完成工作而不辞辛劳地去努力获取测试认证的测试员值得赞扬。但是,目前还没有大量关于经过认证的软件测试员所执行的测试工作和那些没有经过认证的软件测试员所完成的相同种类测试工作之间缺陷去除效率水平的对比经验数据。

警告和注意事项 关于软件测试的主要警告是,测试并不能发现很多错误或缺陷。50多年来,尽管也进行了广泛测试,软件行业还是经常交付带有成百上千潜在缺陷的大型软件应用。

关于测试的第二个警告是,测试不能发现需求错误,比如著名的千年虫问题。一旦需求中出现错误,并且通过使用审查、质量功能展开(QFD)以及一些其他非测试质量方法仍然没有发现该错误,所有将要完成的软件测试都是在确认该错误。这就是正确的需求和设计文档为什么对软件测试的成功至关重要。这也解释了为什么需求和设计文档的正式审查能够提升每一个测试阶段的测试效率约5%。

结论 软件测试文献内容涵盖广泛,但几乎所有的文献都缺乏涉及下列主题的量化数据:缺陷去除效率、测试成本、测试团队人员组成、测试专家、测试投资回报、测试人员的生产率。同时,还有许许多多含有测试信息的书籍和数以百计的测试信息网站。

测试领域有许多非营利性组织,比如软件测试协会(the Association for Software Testing, AST)、美国质量学会(the American Society for Quality, ASQ)以及软件质量全球联合会(Global Association for Software Quality, GASQ)。

很多城市都有本地或者地区级的软件质量组织。很多营利性的测试协会也会举办各种软件测试大会和专题研讨会,以及提供软件测试认证考试。

鉴于过去50多年软件测试在软件工程中的核心作用,测试文献的上述不足出乎意料、令人惊讶。一个技术职业,对于预防和去除严重错误的最高效及最符合成本效益的方法一无所知,没有资格被称为“工程”。

某些最新的测试形式比如测试驱动开发(TDD),通过将测试用例开发移入软件开发周期的早期阶段,并将测试用例与需求和设计结合起来,正在朝着积极的方向发展。这些测试策略的变化带来了更高的缺陷去除效率水平,同时也降低了测试成本。

但是,要以更具成本效益的方式实现高水平的软件质量,单靠测试这一种方式是远远不够的。缺陷预防方法与结合了正式审查、静态分析、自动化测试和手工测试的一整套多阶段缺陷去除活动的协同组合可以提供最佳的整体软件质量结果。

就软件行业整体而言,多年以来,潜在缺陷太高而缺陷去除效率又太低。这种不幸组合提高了开发成本,延长了开发周期,导致了大量项目失败及诉讼案件,还使软件维护和客户支持成本远远高于它本来应该付出的成本。

与2009年相比,诸如团队软件过程(TSP)、质量功能展开(QFD)、软件六西格玛、联合应用设计(JAD)、参与审查以及认证可重用组件等缺陷预防方法具有降低潜在缺陷达80%或者更多的理论潜力。换句话说,潜在缺陷可以从当前的每功能点5.0个缺陷下降到每个功能点约1.0个缺陷甚至更少。

正式审查、静态分析、测试驱动开发、既使用自动化测试又使用手工测试方法以及认证可重用测试用例等缺陷去除活动的组合可以将缺陷去除效率水平从2009年当前的平均大约85%提高到大约97%。在许多情况下,甚至可以实现接近99.9%的高水平。

在2009年,除了一些非常成熟的软件组织,缺陷预防和缺陷去除活动的有效结合完全可以为软件项目所用但实际上很少使用。当前所缺乏的不是这么多可以改善软件质量的技术和方法,而是缺乏对上述最佳组合实际上多么有效的深入认识,缺乏对单单测试这一种方法多么无效的清醒意识,缺乏广泛的软件质量度量方法和质量基准研究,而这恰恰拖了软件质量改进的后腿。

同样颇具价值的还有软件缺陷预测估算工具,这些工具可以预测同行检查、正式审查、静态分析、自动测试阶段和手动测试阶段任意组合的潜在缺陷和缺陷去除效率水平。这些工具在2009年已经存在,已由诸如软件生产力研究所(SPR)、SEER、Galorath及Price Systems等公司推向市场。能够预测潜在软件缺陷对客户造成的各种损失的软件工具更为复杂,但也已经存在该类工具软件的原型了。

最后的结论是,直到软件行业的平均缺陷去除效率水平通常达到95%以上,重要软件应用上达到99%,软件行业甚至不应该假装自己是一个真正的工程学科。没有有效质量控制的“软件工程”就是一个骗局。

5.7.4 软件质量保证(SQA)组织

本书作者曾在IBM位于加利福尼亚Palo Alto和Santa Teresa的软件质量保证组织中工作过5年。因此,笔者可能会有残余偏好,青睐与IBM的SQA组织功能相同的SQA组织。

在软件行业里,SQA组织的职责和功能仍然含混不清。在笔者的客户(主要是《财富》500强企业)里,SQA组织的大概运作情况如下所示:

- 约50%的公司,SQA组织主要是软件测试组织,执行回归测试、性能测试、系统测试和当大型软件系统进行集成时所使用的其他类型测试。SQA组织汇报给一个主管软件工程的副总裁,或者首席信息官(CIO),或者本地开发经理,不是一个独立的组织。这些SQA组织可能有些质量度量的职责,但软件测试是他们工作的主要焦点。这些SQA组织的规模往往相当大,雇用的SQA人员总数可能超过软件工程总人数的25%。
- 约35%的公司,SQA组织是企业软件质量评估与度量以及确保遵守本地及国家质量标准的核心力量。但是,SQA组织是从测试组织中分离出来的,且只从事非常有限且特殊的测试工作,比如标准符合性测试。为了具有独立观点,SQA组织汇报给自己所属的质量副总裁,而不再是开发或者测试组织的一部分。(这是笔者在IBM

SQA 组织工作时 IBM 的 SQA 组织形式。) 这些组织往往规模相当小, 雇用的 SQA 人员数目在软件工程总人数的 1% ~ 3% 之间。

- 约 10% 的企业, 只有软件测试组织, 根本就没有任何 SQA 组织。测试团队通常汇报给首席信息官 (CIO), 或者某个副总裁, 或者资深软件高管。在这种情况下, 尽管也有些许的软件质量度量工作, 软件测试才是他们工作的主要焦点。虽然测试组织可能规模庞大, 但 SQA 的人员编制为零。
- 约 5% 的公司, 有一个 SQA 副总裁和可能一到两个助手, 再无其他 SQA 人员。在这种情况下, SQA 无非是客户来访时进行演出的一场表演。这样的组织可能拥有汇报给各个开发经理的软件测试团队。这些所谓的 SQA 组织只有管理人员而没有 SQA 人员, 其雇员数尚不及软件从业人员总数的千分之一。

由于软件质量保证 (SQA) 涉及的内容比软件测试工作要多得多, 我们可以来看看独立于软件测试组织而独立运作的“传统”SQA 组织的活动和职责。

1. 开发期间及发布之后, 收集和度量软件质量, 包括分析测试结果和测试覆盖率。在诸如 IBM 等某些组织里, 还包括计算缺陷去除效率水平。
2. 为重要的新应用软件预测软件质量水平, 包括构建特殊的质量评估工具。
3. 进行质量统计研究, 或者执行根本原因分析。
4. 检查和培训质量方法, 比如质量功能展开 (QFD) 或软件六西格玛。
5. 作为会议主持或记录员而参与软件审查会议, 对软件人员进行审查方法培训。
6. 确保遵从本地、国家及国际软件质量标准。例如, 在组织实现 ISO 9000 认证方面, SQA 团队的作用非常重要。
7. 监控与软件能力成熟度模型集成 (CMMI) 各个等级相关的活动。在企业软件过程改进及提升到 CMMI 更高等级方面, SQA 团队起到了一个非常重要的作用。
8. 执行诸如标准遵从性等专业性测试。
9. 对新员工进行软件质量相关主题的培训。
10. 从外部组织 (比如国际软件基准组织 (ISBSG)) 获取软件质量基准研究数据。

IBM SQA 团队的一个主要职责是确定新应用软件的质量水平是否足够地好, 是否达到了可以将该应用交付给客户的程度。SQA 组织可以叫停交付那些他们认为不具备足够质量水平的软件。

开发经理可以对 SQA 组织所作出的停止发布有问题软件的决定提出上诉。而这种上诉由 IBM 总裁或者一个资深副总裁进行最终裁决。这种事情不常发生, 但一旦发生, 所有有关方面将会非常认真地对待该事件。SQA 组织被授予这种权力的事实, 对软件开发经理是个强有力的激励, 可以促使他们严肃对待软件质量。

很显然, 要想有权力叫停一个新应用的交付, SQA 组织必须有其自己的行政管理体系及自己独立于软件开发组织的资深副总裁。如果 SQA 报告给一个软件开发高管, 那么, 来自上级的威胁或压力可能会使 SQA 角色无法有效发挥作用。

IBM SQA 组织一个独特的特色是正式“SQA 研究”功能, 它提供时间和资源以从事超越当前可用的最先进 SQA 主题的研究。例如, IBM 的第一个质量评估工具就是在这类研究

计划中开发出来的。研究人员可以为任何感兴趣的课题提交提案,而那些被挑选出来并获得批准的提案将会获得时间和必要的资金以对其进行研究。

一些公司鼓励其 SQA 人员和其他软件工程师撰写技术书籍,或者为外部期刊杂志,比如 CrossTalk (美国空军的软件杂志)或 IEEE 的某些杂志,撰写技术文章。

有一家公司,ITT,作为其软件工程研究实验室工作的一部分,允许其 SQA 人员在工作时间撰写技术文章,甚至还在创作书籍最终完成稿上提供辅助。最重要的一点是,允许作者们保留他们所出版技术书籍的版税收入。

一个有趣的现象是,几乎每一个平均缺陷去除效率水平超过 90% 的公司都有一个正式且活跃的 SQA 组织。虽然正式而活跃的 SQA 团体与质量优于平均水平有关联,但却没有足够的数据可以断言 SQA 是高质量的首要原因。

其原因是,软件质量较差的大多数组织没有任何软件质量度量措施,并且只有在这些组织委托专家对其软件进行专门的质量评估,或者因为其较差软件质量而被起诉并闹上法庭时,他们糟糕的软件质量水平才会暴露出来。

要说“具有正式 SQA 团队的组织其平均缺陷去除效率超过 90%,而没有正式 SQA 团队、开发类似软件的相似公司其平均缺陷去除效率低于 80%”,这话倒是没有说错。但不幸的事实是,只有那些具有正式 SQA 团队的公司才有可能知道他们的缺陷去除效率水平到底是多少。实际上,软件质量度量实践如此糟糕,甚至有些确实具有 SQA 团队的公司也真不知道他们自己的缺陷去除效率水平到底是多少。

人口数据 在软件世界里,SQA 在数量上并不算多,但早已成为软件质量创新的一个重要来源。截至 2009 年,美国可能有 5000 名受雇的全职 SQA 从业人员。

SQA 组织在诸如 SAP、微软、Oracle 等构建系统软件、嵌入式软件或者商业软件的那些公司里非常常见。而在诸如银行、金融等公司的 IT 组织里 SQA 部门并不多见,尽管在规模较大的这类公司里也确实存在。

很多城市有其本地的 SQA 团体,也有很多国家和国际级的质量协会。

在软件应用的 SQA 支持方面有个有趣的反常现象。使用团队软件过程(TSP)的开发团队有他们自己的内部 SQA 性质组织,负责收集大量的缺陷和质量数据。因此,TSP 团队通常不需要任何企业级 SQA 组织参与其中。当然,为了企业报告,他们也会向企业级 SQA 组织提供相关的数据,但他们没有任何参与其中的 SQA 人员。

项目规模 通常,对于规模超过 2500 个功能点的大型应用软件,SQA 的参与是强制性的。而 SQA 的参与对较小的应用软件可能也有帮助,它们往往比大型软件的质量更高。既然 SQA 资源比较有限,专注于大型应用软件可能是使用 SQA 人员的最佳方法。

生产率 对于 SQA 组织,当前并没有有效的生产率数据。但是,一个有趣的重要事实是,那些具有 SQA 团队参与并且设法使其缺陷去除效率达到 95% 以上的软件应用,其生产率常常远远好于那些缺乏 SQA 组织参与的同等级规模应用软件。

即使 SQA 组织自身的生产率比较含糊不清,SQA 团队所支持应用程序的质量和生产率度量结果也表明了 SQA 具有显著的商业价值。

进度 SQA 团队的主要进度问题是他们所支持的应用软件的整体进度。与生产率和软

件质量一样,有证据表明,软件应用项目中 SQA 的存在可以有效预防整体进度延误。

实际上,如果 SQA 能够在项目中成功引入正式审查,有可能缩短项目进度。

缩短软件开发项目进度的最有效方式是,使用缺陷预防方法使得软件只有极少数缺陷,使用预测试审查和静态分析在正式测试开始之前就将仅有缺陷的大多数去除掉。

既然 SQA 团队在缺陷预防和早期缺陷去除上努力推动,一个有效的 SQA 团队将有利于开发进度的正常推进,这对于大型应用软件尤其如此,这些大型软件的开发进度常常落后。

就软件行业整体而言,由于数量过多的软件缺陷而导致的进度延误是软件项目成本超支、进度落后的主要原因,同时也是项目取消的主要原因。有效的 SQA 团队可以最大限度地减少这些普遍存在的问题。

事实证明,有效的 SQA 组织可以显著降低软件项目成本、明显改善项目进度。然而,由于很多公司的高层管理人员并不理解软件高质量的经济价值,常常将高质量视作软件产品可有可无的奢侈品而不是商业上的必需品,因此在经济衰退时期, SQA 人员往往是首当其冲被裁掉的软件从业人员之一。

质量 SQA 团队作用的核心在于软件质量,包括软件质量度量、软件质量预测以及长期软件质量改进。SQA 团队在 ISO 标准及 CMMI 的推广上也有重要职责。SQA 组织也需要教授软件质量课程,在诸如质量功能展开(QFD)及软件六西格玛等方法的开发上提供帮助。事实上,许多 SQA 人员都具有六西格玛黑带认证,这并不罕见。

2009 年的今天,当软件项目团队使用测试驱动开发(TDD)方法时, SQA 团队的职责有着诸多不确定性。因为 TDD 方法相当崭新, TDD 与 SQA 的相互交融仍在不断发展中。

正如已在本章测试部分提到过的,建议管理人员或者有资格签署软件合同的读者将 95% 作为可接受的缺陷去除效率最低水平。每一个外包合同、每一份内部质量计划以及与软件供应商的每一项许可协议,都应该要求提供证据以证明其软件开发组织在缺陷去除效率方面能够高于 95%。

软件行业有一个令人不安的现象,需要进行更多的研究。尽管经过了大量测试,有时还有大型 SQA 团队的参与,规模在 10 000 个功能点以上的大型系统还是经常带着数以百计的潜在缺陷被发布出去。这些大型系统最终很多卷入了官司,而笔者恰好在这样的案例中做过专家证人。在这样的大型系统开发过程中经常发生的事情是, SQA 团队的建议没有被采纳,而在被误导企图压缩项目进度的情况下,项目经理经常对软件项目质量控制敷衍了事。

专业化 SQA 专业涵盖了非常广泛的技能,包括统计分析、功能点分析,也包括软件测试技能。其他特殊技能包括六西格玛、复杂度分析和根本原因分析。

警告和注意事项 关于 SQA 的主要警告是, SQA 是为软件项目提供帮助,而不是来阻碍项目进展的。教条主义的态度将对 SQA 团队与开发团队及测试团队的有效合作产生适得其反的效果。

结论 有效的 SQA 组织不仅有利于提高产品质量,而且对项目进度和成本也颇为有益。不幸的是,在经济衰退期间, SQA 团队是首当其冲被裁员或精简的软件人员之一。随着 2009 年经济衰退的延续,关于 SQA 的未来在美国商界产生了很大的不确定性。

因为高质量有益于软件项目成本和进度，作为 SQA 标准职责的一部分，迫切需要 SQA 团队采取积极行动和措施，其中包括软件缺陷去除效率以及软件质量经济价值的度量措施。如果 SQA 能将更多的正式软件质量基准研究结果应用于公司，收集数据并提交给质量基准研究团体，这些数据将对公司和软件行业均有益处。

有些非营利性组织也从事 SQA 研究，比如美国质量协会（ASQ）、软件质量全球联合会（GASQ）等。

很多城市都有本地或者地区级的软件质量组织。很多营利性的 SQA 协会，比如质量保证研究所（QAI）也举办各种 SQA 大会和专题研讨会，以及提供 SQA 认证考试。

SQA 需要主动提供帮助，在软件项目中引入缺陷预防方法与结合了正式审查、静态分析、自动化测试和手工测试的一整套多阶段缺陷去除活动的协同组合。软件质量没有银弹，但各种质量方法的融合将会对提高软件质量非常有效。SQA 团队是这些有效混合方法信息和培训的最合乎逻辑的最佳来源。

在 2009 年的今天，软件缺陷预防和缺陷去除活动的有效结合已经存在。但除了一些非常成熟的组织，实际上这些有效方法组合很少使用。正如在本书测试部分提到过的，当前所缺乏的不是这么多可以改善质量的技术，而是缺乏对上述最佳组合实际上多么有效的认识，缺乏广泛的质量度量方法和质量基准研究，而这恰恰拖了软件质量改进的后腿。

同样颇具价值的还有软件预测估算工具。这些工具可以预测、正式审查、静态分析、自动测试阶段和手动测试阶段任意组合方法的潜在缺陷和缺陷去除效率水平。通常情况下，SQA 团队应该拥有这些工具并频繁地使用它们。实际上，软件行业的首款软件质量预测工具就是由 IBM 的 SQA 组织于 1973 年在加利福尼亚的 San Jose 开发出来的。

最后的结论是，SQA 团体需要不断推动，直到软件行业的平均缺陷去除效率水平整体达到 95% 以上，重要软件应用上达到 99%。小于这些数据的任何结果都是远远不够和不专业的。

5.8 总结

佛瑞德·布鲁克斯（Frederick P. Brooks），IBM 软件先驱之一，在他的经典著作《人月神话》中观察到，软件强烈地受到其开发组织的组织结构影响。Fred 的书出版后不久，也曾任 IBM 工作过的本书作者指出，大型系统往往被分解以适应现有的组织结构。尤其是，有些主要功能被人为分割以适应标准的 8 人规模的部门。

本书只触及了组织结构问题的皮毛。在小团队和大团队相对优势方面还需要更多的深入研究。此外，8 名雇员汇报给一个经理的“平均”管理跨度很可能需要修正。对各种不同团队规模的有效性研究发现，将管理跨度从 8 人提升到 12 人将使能力不足的管理人员重返技术岗位，并最大限度地减少管理纠纷，而这些管理纠纷往往是长期存在的问题。更进一步，由于软件应用的规模在不断增大，较大的管理跨度可能会更好地匹配当前的软件架构。

另外一个需要更多额外研究的主要议题是那些真正的大型软件团队，包括 500 人或更多的软件人员和几十名软件专家。最有效管理如此大规模多元技能团队的方法的经验数据

十分稀缺。如果这样的团队分散在各地,那么又增加了需要额外研究的另一个主题。

最近,Victor Basili 博士、Nachiappan Nagappan 和 Brendan Murphy 研究了微软的组织结构问题。得出的结论是,Windows Vista 的很多问题都可以追溯到微软的组织结构问题上。

但是,相比于影响软件工程的其他主题,比如方法、工具、编程语言和测试等,当前有关软件组织结构及其影响的文献资料非常稀少。

由于某种程度上经理们倾向于保护自己的势力范围,正式的组织结构往往带有一定领地边界性质。这往往使各个团队的视野比较狭窄。支持跨部门沟通的最新非正式组织形式正越来越受到人们欢迎,这也增加了创新和解决问题的机会。

软件团队的组织结构应该是动态的,随着技术的变化而不断调整。但遗憾的是,现实中的组织结构往往滞后于它们本来应该的样子很多年。

随着 2009 年经济衰退的延续,可能会激发人们进行更多组织结构主题方面的研究。例如,需要仔细考察的新课题包括 Wiki 站点、使用虚拟现实技术进行通信的虚拟部门,以及为减少燃油消耗而出现的家庭办公的有效性等。

一个几乎没有任何文献提及、非常重要的话题是如何以最小破坏性的方式进行裁员或精简机构。这一主题已在本书第 1 章和第 2 章讨论过,但除此之外,可供参考的资料非常少。由于公司往往会裁掉不该裁的人,裁员经常会使之后数年公司的运营效率水平受到影响,甚至一蹶不振。

鉴于软件开发进度缓慢,另一个需要深入研究的重要主题是对各个团队分散在不同时区、彼此相隔 8 小时时差的全球性组织进行研究。这种组织允许软件应用和工作产品在全球范围内从一个团队到另一个团队进行轮班转换,这样,允许软件开发工作 24 小时不间断进行,而不是每天只进行 8 小时开发。

最后一个需要更多深入研究的组织结构议题是能够创建可重用模块及其他可重用交付物的最佳组织。而后,直接从可重用组件构建软件应用,而不再是一行代码一行代码地自己从头开始编写。

参考文献

- Brooks, Fred. *The Mythical Man-Month*. Reading, MA: Addison Wesley, 1995. (最新中文版《人月神话》(32 周年中文纪念版), UML China 翻译组 汪颖译,清华大学出版社 2007 年 9 月出版)
- Charette, Bob. *Software Engineering Risk Analysis and Management*. New York: McGraw-Hill, 1989.
- Crosby, Philip B. *Quality is Free*. New York: New American Library, Mentor Books, 1979. (最新中文版《质量免费》,杨钢、林海译,山西教育出版社 2011 年 5 月出版)
- DeMarco, Tom. *Controlling Software Projects*. New York: Yourdon Press, 1982.
- DeMarco, Tom. *Peopleware: Productive Projects and Teams*. New York: Dorset House, 1999. (最新中文版《人件》(第 2 版), UML China 翻译,清华大学出版社 2011 年 1 月出版)
- Glass, Robert L. *Software Creativity*, Second Edition. Atlanta: *books, 2006.
- Glass, R.L. *Software Runaways: Lessons Learned from Massive Software Project Failures*. Englewood Cliffs, NJ: Prentice Hall, 1998.

- Humphrey, Watts. *Managing the Software Process*. Reading, MA: Addison Wesley, 1989.
- Humphrey, Watts. *PSP: A Self-Improvement Process for Software Engineers*. Upper Saddle River, NJ: Addison Wesley, 2005.
- Humphrey, Watts. *TSP – Leading a Development Team*. Boston: Addison Wesley, 2006. (中文版《TSP—领导开发团队》, 张家才、江贺、车皓阳译, 人民邮电出版社 2007 年 1 月出版)
- Humphrey, Watts. *Winning with Software: An Executive Strategy*. Boston: Addison Wesley, 2002. (中文版《软件制胜之道—执行的策略》, 张明译, 科学出版社 2011 年 5 月出版)
- Jones, Capers. *Applied Software Measurement*, Third Edition. New York: McGraw-Hill, 2008.
- Jones, Capers. *Estimating Software Costs*. New York: McGraw-Hill, 2007. (中文版《软件项目估计》(第 2 版), 刘从越、郝建材、申冬凯译, 电子工业出版社 2008 年 3 月出版)
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Boston: Addison Wesley Longman, 2000. (中文版《软件评估、基准测试与最佳实践》, 韩柯译, 机械工业出版社 2003 年 4 月出版)
- Kan, Stephen H. *Metrics and Models in Software Quality Engineering*, Second Edition. Boston: Addison Wesley Longman, 2003. (最新中文版《软件质量工程的度量与模型》, 王振宇、陈利、余扬译, 机械工业出版社 2003 年 10 月出版)
- Kuhn, Thomas. *The Structure of Scientific Revolutions*. Chicago: University of Chicago Press, 1996. (最新中文版《科学革命的结构》(第四版), 金吾伦、胡新和译, 北京大学出版社 2012 年 11 月出版)
- Nagappan, Nachiappan, B. Murphy, and V. Basili. *The Influence of Organizational Structure on Software Quality*. Microsoft Technical Report MSR-TR-2008-11. Microsoft Research, 2008.
- Pressman, Roger. *Software Engineering – A Practitioner's Approach*, Sixth Edition. New York: McGraw-Hill, 2005. (最新中文版《软件工程: 实践者的研究方法》(原书第 7 版), 郑人杰、马素霞译, 机械工业出版社 2011 年 5 月出版)
- Strassmann, Paul. *The Squandered Computer*. Stamford, CT: Information Economics Press, 1997.
- Weinberg, Gerald M. *Becoming a Technical Leader*. New York: Dorset House, 1986. (最新中文版《成为技术领导者——解决问题的有机方法》, 朱于军、李先华、朱赛春、裘强译, 清华大学出版社 2003 年 9 月出版)
- Weinberg, Gerald M. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971. (最新中文版《程序开发心理学》(银年纪念版), 韩江、陈玉译, 电子工业出版社 2010 年 3 月出版)
- Yourdon, Ed. *Outsource: Competing in the Global Productivity Race*. Upper Saddle River, NJ: Prentice Hall PTR, 2005.
- Yourdon, Ed. *Death March – The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*. Upper Saddle River, NJ: Prentice Hall PTR, 1997. (最新中文版《死亡之旅》(原书第 2 版), 周浩宇、杨华译, 机械工业出版社 2012 年 1 月出版)

项目管理和软件工程

6.1 引言

项目管理是软件工程各环节中最薄弱的环节,也是很多大学学历教育课程中的薄弱环节之一。与糟糕的编程工作和拙劣的软件工程实践相比,越来越多的软件问题以及诸如成本超支、进度落后等项目问题都可归咎于软件项目管理不善。项目管理不善对软件工程的害处之大,完全值得用整整一本书来论述软件项目管理的最佳实践。

笔者曾在大量涉及项目取消、软件质量问题以及其他重大失败的诉讼案件中担任专家证人。几乎在每一个诉讼案件中,笔者都看到了糟糕的项目管理实践。不但项目管理问题非常常见,而且在很多诉讼案件中,项目经理及高层管理人员错误地认为取消正式审查和缩减测试工作等措施将会缩短软件开发周期,实际上这严重干扰了优秀软件工程实践的实施。

例如,在大量违反合同案件中,诸如估算不足、质量控制不足、变更控制不足以及误导性甚至虚假的状态跟踪等项目管理问题频繁发生。

随着经济衰退的继续,在不降低运营效率的前提下,为降低项目成本,认真分析软件工程的每一个方面正变得越来越重要。改善项目管理正是成功降低项目成本的关键举措之一。

在软件环境中,需要重新定义“项目管理”。术语“项目管理”所涵盖的范围已经被项目管理工具厂商人为地缩小了,当前的“项目管理”只局限于关键路径分析及制定各种项目进度辅助计划(诸如 PERT[⊖]图及甘特图)等项目活动。而对于成功的软件项目管理,还必须支持其他很多项目活动。

根据对近 150 家公司观察到的实际情况,表 6-1 说明了当前存在的 20 种核心项目管理功能以及它们的执行情况。得分范围为从 -10 到 +10,其中 -10 代表相应的项目管理功能实际状况非常糟糕,+10 代表相应的项目管理功能实际表现非常卓越。

这种从 +10 到 -10 的评分方法,其中间值或者平均值为 0。过去多年的观察结果表明,在太多关键项目管理活动上,项目管理水平远远低于平均水平。

⊖ PERT 图,即 Program/Project Evaluation and Review Technique 图,计划评审技术。一种利用类似流程图的箭线图,描绘出项目包含各种活动的先后次序,标明每项活动的时间或成本,制定项目计划及对计划予以评价的技术。——译者注

表 6-1 的第一项, 报告“红灯”项目问题^①, 是指通知客户或高层经理们某个项目有了大麻烦。在几乎每个软件项目违约诉讼中都存在项目问题被刻意隐瞒或忽略的现象, 因此错过了解决问题的最佳时机, 直到这些问题变得太严重而再也无法解决。

表 6-1 2009 年软件项目管理情况

项目管理功能	分数	定义	项目管理功能	分数	定义
1 报告“红灯”项目问题	-9.5	非常糟糕	12 状态和进度跟踪	2.0	微弱
2 缺陷去除效率度量	-9.0	非常糟糕	13 成本估算	3.0	一般
3 项目完成时的基准评估	-8.5	非常糟糕	14 价值评估	4.0	一般
4 需求变更评估	-8.0	非常糟糕	15 质量度量	4.0	一般
5 项目完成时的事后分析	-8.0	非常糟糕	16 过程改进规划	4.0	一般
6 质量评估	-7.0	非常糟糕	17 质量和缺陷跟踪	5.0	良好
7 生产力度量	-6.0	糟糕	18 软件评估	6.0	良好
8 风险评估	-3.0	糟糕	19 成本跟踪	7.0	优秀
9 过程改进跟踪	-2.0	糟糕	20 挣值跟踪	8.0	优秀
10 进度估算	1.0	微弱	平均	-0.8	糟糕
11 初始应用规模估算	2.0	微弱			

软件项目经理表现平平, 其主要原因可能在于美国大学和研究生教育缺乏有效的项目管理课程。几乎没有多少软件工程师及更少 MBA 学生学习过软件质量的经济价值以及如何度量缺陷去除效率水平等方面的任何知识和课程, 而实际上缺陷去除效率水平是软件工程中最最重要的一个度量指标。

如果我们仔细检查 2009 年可用的有效软件项目管理工具和方法, 且给予软件项目经理最先进水平的项目管理培训, 一个截然不同的项目管理面貌将会呈现在我们面前。

表 6-2 假设, 在 10 年之内, 将会有一门面向项目经理、极大改善了的项目管理课程可用, 以及项目经理将配备现代化的软件规模估算、成本估算、质量评估和度量工具。该表展示了如果软件项目经理训练有素(接受上述良好的培训课程)且装备精良(配备上述现代化项目管理工具), 项目状况将会表现如何。

与没有进行充分项目估算和质量规划就盲目地启动软件项目不同, 如表 6-2 所示, 软件项目经理对项目进行高精度的计划和估算, 对项目状况进行更精确的度量, 每一个重要应用软件完成时都要创建相应的项目基准, 至少在理论上这是可行的。遗憾的是, 项目管理的日常实际表现要比理论上的软件项目管理技术差得多。

截至 2009 年, 笔者估计只有约 5% 的美国软件项目在项目完成时创建了该项目的生产力和质量数据基准, 而会向某个由诸如国际软件基准组织 (ISBSG)、软件生产力研究所 (SPR)、David Consulting Group、质量与生产力管理集团 (QPMG) 或类似组织所维护的正式基准库提交项目基准数据的软件项目则更是少于 0.5%。

① 在很多软件项目中, 项目整体或某一事项的状态用“绿灯或绿旗”、“黄灯或黄旗”和“红灯或红旗”表示。“绿灯或绿旗”表示状态正常、进展顺利; “黄灯或黄旗”表示有问题但问题不严重且处于受控状态; “红灯或红旗”表示问题严重且可能即将或已经失控, 需要优先解决。——译者注

表 6-2 到 2019 年时潜在的软件项目管理情况

项目管理功能	分数	定义	项目管理功能	分数	定义
1 报告“红灯”项目问题	10.0	卓越	12 质量评估	8.0	优秀
2 项目完成时的基准评估	10.0	卓越	13 初始应用规模估算	8.0	优秀
3 项目完成时的事后分析	10.0	卓越	14 成本估算	8.0	优秀
4 状态和进度跟踪	10.0	卓越	15 风险评估	7.0	良好
5 质量度量	10.0	卓越	16 进度估算	7.0	良好
6 质量和缺陷跟踪	10.0	卓越	17 过程改进跟踪	6.0	良好
7 成本跟踪	10.0	卓越	18 价值评估	6.0	良好
8 缺陷去除效率度量	9.0	优秀	19 过程改进规划	6.0	良好
9 生产力度量	9.0	优秀	20 需求变更评估	5.0	良好
10 软件评估	9.0	优秀	平均	8.4	优秀
11 挣值跟踪	9.0	优秀			

每一个重大软件项目在项目完成时都应该收集该项目的正式基准数据。还应该对这些项目的开发方法进行事后回顾分析以弄清该项目的方法改进是否对未来项目有所帮助。

截至 2009 年，市面上存在很多可用的独立软件项目管理工具，但每一款工具都只支持所有软件项目管理任务的某些部分。新一代集成软件项目管理工具即将推向市场，该工具承诺消除现存项目管理工具的不足，改善各个工具之间彼此分享信息的能力。新型项目管理工具（比如项目方法管理工具）也已被软件项目管理社区加入到其可用的项目管理工具集中。

软件项目管理是 21 世纪要求最为苛刻的工作之一。软件项目经理所负责的是构建公司历史上曾经努力构建过的最昂贵公司资产。例如，大型软件系统的开发成本远比建造公司所在办公大楼的费用要多得多，其所需开发时间也远比建造该办公大楼所需时间长得多。事实上，规模在 100 000 个功能点数量级的大型软件系统的成本比建造一个圆顶足球场、一栋 50 层摩天大楼或者一艘 70 000 吨邮轮的花销还要多得多。

大型软件系统不仅成本昂贵，而且还是人类历史上失败率最高的人造产品之一。术语“失败”指软件开发项目因成本超支、进度落后而没有完成即被取消，或者项目进度落后于计划进度超过 25% 以上。

对于软件故障或者灾难，绝大多数指责都指向了项目管理团队而非技术团队。表 6-3 来自笔者很久之前所写的一本书《软件系统失败与成功模式》（Patterns of Software System Failure and Success），由国际汤姆森出版社（the International Thomson Press）出版。需要注意的是，软件项目经理在成功项目与和项目取消及严

表 6-3 成功和不成功项目的软件管理情况对比

项目活动	成功项目	不成功项目
规模估算	良好	糟糕
项目规划	优秀	一般
项目评估	优秀	非常糟糕
项目跟踪	良好	糟糕
项目度量	良好	非常糟糕
质量控制	卓越	糟糕
变更控制	卓越	糟糕
问题解决	良好	糟糕
风险分析	良好	非常糟糕
人事管理	良好	糟糕
供应商管理	良好	糟糕
整体情况	优秀	糟糕

重超支有关的失败项目上的表现对比。

正如第5章提到的,笔者在项目失败和违反合同等软件诉讼分析中所做研究可以得出这样的结论:与软件项目中的软件工程师数量相比,项目失败与参与软件项目的管理人员数目更加紧密相关。

包括6名以上一线经理的软件项目经常进度落后、成本超支。而牵涉了超过12名一线经理的软件项目往往进度严重落后,常常会被取消。

由此可以很容易看出,软件项目管理功能的缺失是软件灾难的最根本原因。相反,卓越的项目管理比几乎任何其他因素(比如购买更好的工具、改变编程语言等)更能提高项目成功的概率。(该结论对1000个功能点以上的大型应用软件项目也成立。而对于规模在100个功能点数量级的小型应用软件项目,软件工程师的个人技术技能仍然是项目成功与否的决定性因素。)

整体上,提高软件项目管理绩效比任何其他已知的项目活动更能最大限度地提高软件项目成功的可能性、最大可能地减少项目失败概率。然而,提高软件项目管理绩效也是更困难的软件改进策略之一。如果容易做到的话,软件行业应该具有比现实情况多得多的成功项目、少得多的失败项目。

大多数软件项目的失败都可归因于项目管理的失败而不是技术人员的失误。例如,在因进度落后、成本超支而导致项目取消的所有项目中,超过70%的此类失败项目都与低估项目进度和资源需求有关联。另一个常见的项目管理问题是忽视或低估与质量控制及缺陷去除相关的项目工作。再一个常见的项目管理问题就是无法有效处理需求变更。

鉴于软件系统建造的高昂成本和巨大难度,你可能会认为软件项目经理应该训练有素,装备有最先进的项目规划和估算工具,拥有大量历史软件项目成本结构的实证分析,熟悉非常全面的风险分析方法。但这些都是我们做出的自然而然的假设,而现实情况并非如此。表6-4说明了在进度提前、正好、落后的软件项目上项目管理工具的使用情况。

表6-4说明,在进度提前的软件项目上,项目经理不但使用了多种多样的项目管理工具,而且还使用了这些工具的更多功能。

表6-4 项目管理工具的数量和规模范围(规模数据以功能点指标表示)

项目管理	落后	正好	提前	项目管理	落后	正好	提前
项目规划	1000	1250	3 000	资源跟踪	300	750	1500
项目成本估算			3 000	价值分析		350	1250
统计分析			3 000	成本差异报告		500	1000
方法管理		750	3 000	人员支持	500	500	750
项目基准			2 000	里程碑跟踪		250	750
质量评估			2 000	预算支持		250	750
评估支持		500	2 000	功能点分析		250	750
项目度量			1 750	逆火分析: LOC 到 FP			750
项目组合分析			1 500	功能点小计	1800	5350	30250
风险分析			1 500	工具数量	3	10	18

而对于进度落后的软件项目，部分原因是软件项目经理学术素养不足。大多数软件项目经理要么完全缺乏软件项目管理的培训，要么充其量只在手头工作方面接受过极其有限的部分培训。更糟糕的是，软件项目经理经常严重缺乏最先进的软件项目管理工具。

根据笔者从事咨询研究所收集到的数据，曾接受过软件成本估算、项目规划或风险分析等方面正式培训的美国软件项目经理少于 25%；曾接触过现代软件成本估算工具的美国软件项目经理少于 20%；而访问过大量与他们所负责项目类似软件项目的已验证历史数据的项目经理则少于 10%。

项目经理缺乏足够的培训和必要的工具将使软件项目麻烦不断。当前市场上至少有数十种广泛使用的软件成本估算工具，比如 COCOMO、KnowledgePlan、Price-S、SEER、SLIM 等。在众多软件项目基准数据来源中，国际软件基准组织（ISBSG）拥有最多可用的基准数据收集。

相比之下，尽管在某些主题上（如软件质量控制和软件安全）仍然存在很大不足，设计和构建软件的技术人员通常在分析、设计和软件开发方面受到过相当良好的专业培训。

近些年来，短语“项目管理”的涵义不幸地为了用于支持现实项目经理的自动化软件项目管理工具厂商而做了缩小和错误定义。原始宽泛的项目管理概念包括产生项目成果所需的所有项目活动，这些项目活动包括：可交付物的规模评估、成本估算、项目进度和里程碑规划、风险分析、项目跟踪、技术选择、备选方案评估以及项目成果度量等。

当今项目管理工具厂商所使用的“项目管理”概念范围更为狭窄，仅限于与关键路径分析、工作分解结构等项目活动的工作机制和 PERT 图、甘特图以及其他与可视化项目进度辅助手段创建有关、相当有限的一组项目管理功能。这些功能当然是项目经理需要执行的一部分工作，但它们既不是仅有的项目活动，也不是软件项目中最重要项目活动。

对于软件相关项目，传统项目管理工具的不足和狭隘的功能定义更是令软件项目麻烦重重。考虑一个软件项目相关的、非常常见的项目管理问题：当采用一种新的软件开发方法，如敏捷开发或者团队软件工程（TSP）时，对于项目的进度和成本，会有什么结果？

某些商业软件估算工具可以预测敏捷和 TSP 开发方法的结果，但是当涉及备选的软件开发方法时，甚至没有任何一款标准的项目管理工具（比如微软的 Microsoft Project）具有内置的、自动调整其假定的项目开发方法的能力。

对于其他与软件相关的技术，同样也是如此，比如项目管理涉及的正式审查、软件测试、静态分析、ISO 9000 ~ 9004 标准、SEI 成熟度模型、可重用组件、ITIL 等。

本章的关注重点主要是软件项目管理相关的活动和任务。项目经理还需要花费相当多的时间去处理人事问题，比如招聘、绩效评估、加薪以及员工的职业规划等。由于经济衰退，项目经理可能还将面临做出裁员或机构精简的艰难决定。

大多数软件项目经理还会参与部门或公司层面的管理事务，比如制定预算、处理出差申请、规划员工培训以及制定办公室空间使用计划等。对于部门或公司而言，这些都是很重要的管理活动，但是这些活动都不在项目经理职责范围之内，尤其是当项目经理参与项目管理时。

本章的重点是软件项目经理日常关注的项目管理工具和方法，即规模估算、成本估算、

项目规划、项目度量和指标、质量控制、过程评价、技术选择及过程改进等。

项目经理需要了解和熟悉 15 个基本项目管理主题，且每一个主题对专业的软件项目经理来说都非常重要：

1. 软件规模估算
2. 软件项目评估
3. 软件项目规划
4. 软件项目方法选择
5. 软件项目技术和工具选择
6. 软件质量控制
7. 软件安全控制
8. 软件项目供应商管理
9. 软件项目进度与问题跟踪
10. 软件项目度量与指标
11. 软件项目基准
12. 软件项目风险分析
13. 软件价值分析
14. 软件项目过程评价
15. 软件项目过程改进

这 15 个活动并不是需要项目经理关注的全部主题，但就重大软件项目的控制能力而言，它们都是关键主题。

由于笔者之前的书《软件项目估计》（《Estimating Software Costs》，McGraw-Hill, 2007）和《Applied Software Measurement》（McGraw-Hill, 2008）已涉及了众多管理主题，本书将仅涵盖上述 15 个主题中的 3 个：

1. 软件规模估算
2. 软件项目进度与问题跟踪
3. 软件项目基准

规模估算是项目评估的前提条件。规模估算有很多种方法，过去一年内，人们又开发出了很多新的规模估算方法。

软件进度跟踪是所有软件项目管理活动中最关键的活动之一。不幸的是，从很多软件诉讼案件中所披露的证词和文件看，软件项目进度跟踪的实际执行情况实在令人不敢恭维。更糟糕的是，当软件项目陷入麻烦时，糟糕的进度跟踪往往掩盖了真实的项目问题，直到这些问题被发现时已经太晚而无法解决。

软件基准在软件文献中报告很少。当本书写作时，国际标准化组织 ISO 正在起草一项关于软件基准的 ISO 标准。讨论如何收集基准数据以及什么样的报告可以组成有效基准似乎正当其时。

6.2 软件规模估算

术语“规模估算”指预测各种项目可交付物（比如源代码、规格说明书及用户手册）具体数量的方法。软件错误和缺陷也应该包含在规模估算中，因为它们比任何其他项目“可交付物”耗费了更多的金钱和时间。软件缺陷是软件项目的意外可交付物，不管你喜欢与否，它们总会存在，因此软件缺陷也应该包含在规模估算中。由于需求并不总是一成不变的，在开发期间需求也会增长，因此还应该估算应用软件需求变更和增长的规模。

规模估算是项目成本估算的先驱条件，也是最关键的软件项目管理任务之一。规模估算主要关注主要种类的软件可交付物数量的预测，包括但不限于表 6-5 中所示的条目。

由表 6-5 的可交付物列表可见，术语“规模估算”囊括了相当多的可交付物。软件项目中需要预测比源代码多得多的东西，以清晰了解其完整规模和成本估计。

表 6-5 需要量化其规模的软件可交付物

纸质文档	
需求	文本需求 功能性需求（应用的功能特性） 非功能性需求（质量和约束） 用例 用户故事 需求变更（新功能） 需求波动（churn）（不影响整体规模的需求变更）
架构	外部架构（SOA、C/S 等） 内部架构（数据结构、平台等）
规格说明和设计	外部 内部
项目规划文档	开发计划 质量计划 测试计划 安全计划 市场营销计划 维护和支持计划
用户手册	参考手册 维护手册 外国语言翻译
教程材料	
教程材料的翻译	

(续)

	在线帮助的截屏
	帮助截屏的翻译
源代码	
	新开发的源代码
	来自已认证来源的可重用的源代码
	来自未认证来源的可重用的源代码
	继承或遗留应用源代码
	为支持需求变更或波动而添加的代码
测试用例	
	新编写的测试用例
	可重用的测试用例
错误或缺陷	
	需求缺陷(初始)
	需求缺陷(变更需求中的)
	架构缺陷
	设计缺陷
	代码缺陷
	用户文档缺陷
	“不良修复”或次生缺陷
	测试用例缺陷

需要注意的是,虽然软件错误或缺陷是软件项目的意外可交付物,但在大型软件应用中总是存在潜在缺陷,并且这些潜在缺陷会造成极其严重的后果。因此,评估潜在缺陷和缺陷去除效率水平成了软件应用规模估算的关键任务之一。

下面将讨论几种软件应用规模估算的方法,包括但不限于:

1. 具有类似项目的传统类比规模估算法
2. 使用“代码行”指标的传统规模估算法
3. 使用“故事点”指标的规模估算法
4. 使用“用例”指标的规模估算法
5. 使用“IFPUG 功能点”指标的规模估算法
6. 使用其他种类功能点指标的规模估算法
7. 使用功能点粗略估计的高速规模估算法
8. 使用逆火分析的遗留应用高速规模估算法
9. 使用模式匹配的高速规模估算法
10. 软件应用需求变更的规模估算

精确的成本估算和准确的进度规划依赖于正确无误的规模信息,所以规模估算已成为成功软件项目的关键主题。规模和规模变化对于软件项目如此重要,在过去的几年里已经出现了一个被称为“范围经理”的新管理职位。

近来,软件行业开发出了几种新的正式规模或范围控制方法。有意思的是,其中最常见的两种范围控制方法是在彼此非常遥远的地方开发出来的。称为“南方范围控制法”的方法是在澳大利亚开发出来的,而同时被称为“北方范围控制法”的方法则在遥远的芬兰被开发出来。这两种方法均聚焦于需求变更控制,包含了正式规模估算、变更评审以及其他用于量化需求增长和需求变更的技术。尽管已经存在很多种规模控制方法,但南方范围控制法和北方范围控制法似乎均比其他普通方法更为有效。

由于当前存在成千上万的各种类型软件应用,对现存应用软件进行法医鉴定式的仔细分析应该是预测未来应用软件规模的一种不错方法。截至2009年,许多“新”开发应用都只能算是现存遗留应用软件的替代品。因此,如果历史数据可靠而精确的话,那这些历史数据应该非常有用。

软件规模是评估人员编制、项目进度、总工作量、项目成本和软件质量的非常有用的先驱条件。然而,软件规模并不是需要知道的唯一因素。让我们考虑一个建造住宅的比喻。你需要知道房子的平方英尺数或平方米数以便进行成本估算,但是你还还需要知道房子的具体位置、需要使用的建筑材料以及可能需要额外费用的本地建筑法律法规(比如防风飓风窗户或特殊的化粪池系统等)的有关规定。

例如,一座使用普通建筑材料建造、建在平坦郊区、面积达3000平方英尺的普通房子,其建筑成本可达每平方英尺100美元,即总共300 000美元。但是,一座使用进口硬质木材、建在陡峭山坡上需要特殊支撑、面积达3000平方英尺的豪华别墅,其成本则高达每平方英尺250美元,即总计750 000美元。

类似逻辑也同样适用于软件行业。医疗设备中嵌入式软件的成本可能是处理商业数据的同等规模大小应用软件开发成本的两倍多。这是因为,与普通商业应用软件相比,医疗设备中的嵌入式软件的可靠性需要更多的验证和确认工作。

(特别指出:2009年经济衰退之前,建在偏远湖边的一座豪华别墅远离文明世界如此之远,可能需要配备私人机场和自己的发电设施。而这座豪华别墅还可能配有艺术家或手工艺师傅现场专门手工打造精美壁画或特色窗户。这样的建筑,其预算可能高达4千万美元,即超过每平方英尺6000美元。不用说,如果这栋豪宅的主人是位金融家,这座建筑一定是在华尔街崩溃之前建造的。)

软件规模估算长期饱受3个严重问题困扰:(1)大多数时候,很早就需要进行首次成本估算,但软件可交付物的精确规模信息往往在此之后才能确定。(2)某些规模估算方法(如功能点分析)非常费时费力且成本昂贵,大大限制了其在大型应用软件上的使用;(3)软件可交付物的规模往往不是一成不变的,在开发期间往往会有所增长。而规模估算技术常常遗漏这些估算规模的增长和变化。现在,让我们来看看当前常用的几种软件规模估算方法。

6.2.1 传统的类比规模估算法

软件项目规模估算的传统方法是新项目与已完成的旧项目进行类比,因为这些已完

成项目的可交付物规模信息是已知的。然而,当前也有一些新的规模估算方法可用,我们将会在本章稍后详细讨论它们。

传统的类比规模估算法在实践中并不是很成功,其中有很多原因。该方法只能应用于存在类似项目、常见类型的软件项目上。例如,对于软件编译器,类比规模估算法相当有效,因为为数以百计的软件编译器可以拿来做类比。而对于软件人员比较熟悉的其他应用软件类型,比如会计系统、工资计算系统及其他常见的应用软件类型,类比估算法也能很好地工作。但是,如果某个软件非常独特,没有已经构建好的相似类型软件,那么类比规模估算法将毫无用途。

由于很多古老的遗留应用软件在故事点、用例点或者有时甚至是功能点广泛使用之前就已开发出来,因而在为新应用软件提供规模参考方面,并不是每一款遗留应用都能有所帮助。超过90%的遗留应用,无法精确地知道其规模信息。由于“死代码”和外部例程调用的存在,甚至都无法精确地知道其代码数量。同样,这些遗留应用的很多可交付物(比如需求文档、功能说明书、项目计划等)要么早已遗失,要么年久失修,因而它们的规模信息可能无法知晓。

由于遗留应用软件的规模通常以每年约8%的比率增长,因而这些遗留应用当前的规模大小并不能代表其首次发布时的初始规模。有关需求增长的数据记录极其稀少,所以,这使得类比规模估算法完全无用。

更糟糕的是,很多所谓的遗留应用“历史数据”极不精确,不能拿来可靠地预测未来应用软件的规模。即使知道遗留应用的规模大小,为构建遗留应用软件而付出的辛苦努力和成本信息常常也并不完整。历史数据的遗失部分包括无偿加班(几乎从未认真度量过)、项目管理工作以及不算开发团队正式成员的兼职专家(数据库管理员、技术作家、质量保证人员等)的工作。遗留应用软件的额外工作、人员编制和成本信息等数据遗失,在笔者的书中称为“项目数据遗失”。对于只有一两名开发人员的小型应用软件,历史数据的这种遗失微乎其微。而对于拥有几十名团队成员的大型应用软件,项目工作和成本数据的遗失能够达到总项目工作的50%以上。

对于以成本中心模式运作的组织所开发的公司内部软件,项目工作和成本数据遗失情况更为糟糕,因为他们没有任何极其强烈的业务需求要求必须精确地记录项目工作和成本数据。外包的应用软件和根据合同构建的软件在积累项目工作和成本数据方面更为精确些,但即便是这些应用项目也常常遗漏无偿加班。

想想如下这个有趣的事情:IT项目似乎比系统或嵌入式软件具有更高的生产率,其原因之一是IT项目的历史数据遗失比系统或嵌入式软件要多得多。就功能点而言,这种遗失本身就足以使IT项目看起来比同等规模的系统或嵌入式软件生产率至少高15%。原因是,大多数IT项目都是在成本中心环境中开发的,而系统和嵌入式软件则是在利润中心环境中开发的。

国际软件基准组织(ISBSG)的出现在某种程度上使这种情况大大改观,因为ISBSG现已拥有近5000个可为软件工程社区所用的各类应用软件的数据。笔者强烈要求与软件相关的所有读者考虑收集并向ISBSG提交软件项目基准数据。即使因为所有权或商业原因而

不能向 ISBSG 提交这些数据, 内部保留这些数据对公司来说也很有价值。

ISBSG 问卷调查帮助收集数以百计应用程序的相同种类信息, 使规模估算工作更加容易使用这些数据。同样, 那些向 ISBSG 提交项目数据的公司通常具有优于平均水平的项目工作和成本跟踪方法, 因此, 它们的数据很可能比普通公司的数据更加精确。

其他基准组织, 比如软件生产力研究所 (SPR)、质量和生产力管理集团 (QPMG)、David Consulting Group 及许多其他类似组织, 拥有近 60 000 个项目的历史数据, 但这些数据只针对特定客户提供非常有限的分发。这些私有项目数据还比 ISBSG 的数据更为昂贵。根据需要考察的项目数目多少, 与类似相关项目进行对比的一套私人委托基准数据可能花费在 25 000 美元到 100 000 美元之间。当然, 现场私有基准数据相当详细, 同时更正了常见的错误和不足, 因此该数据相当可靠。

对软件行业真正有用的是大量扩展的软件生产力和质量基准数据集。理想情况下, 所有开发项目及所有重大维护和功能增强项目都需要收集足够的项目数据, 这样, 基准数据收集将成为软件行业的标准实践活动, 而不是例外活动。

对于正在进行开发的软件项目, 在展现到目前为止项目中所发现的软件缺陷、花费的项目资金以及确保项目进度进展正常等方面, 项目基准数据非常有价值。实际上, 类似但不太正式的项目数据对于项目状态会议非常有必要, 所以完全有可能出现如下情况: 当项目完成时, 无论是出于基准研究的目的保留还是不保留这些信息, 所需的这些信息都已在项目过程中收集过了, 因此, 正式基准数据的收集几乎不需要任何额外工作。

不幸的是, 尽管类比规模估算法非常有用, 但软件度量实践的缺陷和不足使类比估算法和行业历史数据的价值在很多情况下令人置疑。

类比规模估算法耗时情况 如果有基准数据或者来自类似项目的历史数据, 这种规模估算方法可以较早完成, 甚至在完全知道新应用程序的需求之前就可以开始类比规模估算。该方法是最早的规模估算方法之一。但是, 如果历史数据遗失, 那么类比规模估算方法根本就无法起作用。

类比规模估算法的使用情况 理论上讲, 至少 300 万个现存软件应用可用于类比规模估算。但是, 从对许多大型公司和政府机构的走访情况看, 据笔者推测, 具有足够历史数据的现存遗留应用软件少于 100 000 个, 它们用于类比规模估算非常有用且相当精确。另外大约 100 000 个遗留应用只有部分数据, 但由于这些数据中有太多错误, 用于类比规模估算的话可能会很危险。而剩余 280 万遗留应用要么仅有极少历史数据, 要么根本没有任何历史数据, 要么历史数据太不准确而无法实际使用。对于很多遗留应用软件, 根本没有以任何指标度量的可靠规模数据可用。

进度和费用 如果基准数据或历史数据可用, 那这种规模估算方法快捷而廉价。如果既没有类似项目的规模信息又没有任何历史数据, 则不能使用类比规模估算方法。通常, 来自外部 (比如 ISBSG、David Consulting Group、QPMG 或 SPR) 的基准数据比内部数据更为准确。原因是这些外部基准研究机构会尽力更正基准数据中存在的常见错误, 比如无偿加班数据的遗失等。

警告与注意事项 主要注意事项是, 如果既没有类似项目的历史数据又没有精确的基

准数据, 类比规模估算法根本就无法工作。关于该方法值得注意的是, 类似项目的历史数据常常并不完整, 有可能会遗失某些关键信息, 比如无偿加班。由 ISBSG 或其他一些基准公司收集的正式基准数据通常比大多数公司自己的内部历史数据要更加准确, 那些公司内部数据的可靠性经常非常糟糕。

6.2.2 基于代码行指标的传统规模估算法

20 世纪 60 年代早期, 当“代码行”(LOC)指标出现的时候, 软件应用的规模都很小, 编码工作占到整个软件项目工作的 90%。而 2009 年的今天, 应用软件通常都比较庞大, 编码工作往往不到整个软件开发项目工作的 40%。从 20 世纪 60 年代到现在, LOC 指标就已从对软件项目非常有用途逐渐退化, 直到目前实际上非常有害。当前, 基于 LOC 指标的规模估算方法几乎就是失职。下面将详细阐述为什么 LOC 指标如此有害。

LOC 指标有害的第一个原因是, 经过近 60 年的使用之后, 仍然没有源代码的标准计数规则。LOC 指标既可以用源代码物理行数也可用逻辑语句数来计算。当计算方法在物理行数和逻辑语句数之间进行转换时, 从相同代码段所计算出来的规模可能会出现超过 500% 的差异。

在笔者 1991 年出版的书籍《Applied Software Measurement》第一版中, 包含了基于逻辑语句计算源代码数量的正式规则。这些规则被软件生产力研究所 (SPR) 用于收集软件项目基准数据的逆火 (backfiring) 分析。但 1992 年, 软件工程研究所 (SEI) 发布了他们自己的源代码计算规则, 而这些规则基于物理行数进行计算。由于 SPR 的计数规则和 SEI 的计数规则都被软件行业广泛使用, 但却完全不同, 其实际效果基本上就相当于没有任何计数规则。

(笔者曾做过一项关于主流软件杂志所使用的代码计数方法的研究, 这些主流软件杂志包括《IEEE Software》、《IBM Systems Journal》、《CrossTalk》、《Cutter Journal》, 等等。大约三分之一的文章使用物理行数, 三分之一的文章使用逻辑语句, 而剩余三分之一的文章使用了 LOC 指标却没能 在文章中提及到底是使用了物理行数还是逻辑语句或者两者都有。这是这些软件工程杂志文章作者和审稿人的严重失误。几乎没有人会希望诸如《科学》或《科学美国人》等杂志发表关于某些事物的量化数据却不详细解释用于收集和分析这些结果数据的度量指标。然而, 对于软件工程类期刊, 糟糕的数据度量已司空见惯而非个例。)

LOC 指标有害的第二个原因是, 该指标对高级编程语言的损害与该语言的能力成正比。换句话说, 使用 LOC 指标表示生产力和质量数据, 汇编语言看起来比 Java 或 C++ 更好。

这种损害的出现是因为一个广为人知的制造经济学规律, 但软件社区却不能很好地理解它: 当制造过程包含大量固定成本而制造的产品单元数量下降时, 平均每个产品单元的成本一定会上升。

第三个原因是, LOC 指标无法用于估算或度量软件项目的非编码活动, 比如需求、架构、审计和用户文档。一款由 C 编程语言编写的应用软件的代码量可能是由 C++ 语言编写的同样应用软件代码量的 2 倍之多, 但需求和功能说明书则可能是同等规模。

即使在不改变编程语言等级的情况下, 也不可能从源代码数量就估算出书面文档的规

模。对于诸如 Visual Basic 等没有可用的源代码计数规则的编程语言,几乎不可能预测源代码的规模,更谈不上任何其他可交付物的规模了。

LOC 指标有害的第四个原因是,截至 2009 年,软件行业存在有超过 700 种编程语言,而这些编程语言从非常低级的编程语言(比如汇编语言)到非常高级的编程语言(比如 ASP.NET)各不相同。其中超过 50 种编程语言根本就没有已知的源代码计数规则。

第五个原因是,大多数现代应用软件都用多种编程语言编写,有些应用软件使用了多达 15 种编程语言,而这些语言每个都有自己独特的代码计数规则。即使是 Java 和 HTML 的简单混合也使代码计数变得十分困难。

从历史上看,Visual Basic 语言和它的许多竞争者及其派生语言的出现改变了现代程序的开发方式。尽管“可视化”语言确实具有某些过程式源代码成分,但是很多复杂编程使用的按钮控件、下拉式菜单、可视化表单及可重用组件则要多得多。

换句话说,在没有使用任何可以识别为用于规模估算、度量或者软件评估的“代码行”编程元素的情况下,编程工作就完成了。截至 2009 年,大约 60% 的新应用软件使用面向对象语言或者可视化语言(或者两者都有)开发。实际上,有时在同一款应用软件中可能使用了多达 12 ~ 15 种不同的编程语言。

对于大型系统,在最昂贵项目活动中编程工作本身仅排名第四。比编程工作成本更高的其他 3 项项目活动无法使用代码行指标进行度量或估算规模。同时,排名第五的软件项目主要成本因素——项目管理——也无法很容易地使用 LOC 指标进行估算和度量。表 6-6 给出了大型应用项目中主要软件成本因素的降序排名。

“代码行”度量指标的使用是理解软件项目经济性的一个重大障碍,因为该指标只能度量和评估软件项目五大成本因素中的一个因素。

表 6-6 大型系统软件项目主要成本因素排序

1	缺陷去除(审查、静态分析、测试、发现和修复缺陷)
2	撰写书面文档(各种计划、架构、功能说明、用户手册)
3	会议和沟通(客户、团队成员、管理层)
4	编程
5	项目管理

下面的例子摘录自笔者出版的书籍《Applied Software Measurement》第三版(McGraw-Hill, 2008),这些例子说明了 KLOC 指标在经济性方面的谬误。这里有两个研究案例,分别说明了使用两种编程语言(基本汇编语言和 C++ 语言)编写相同的应用软件分别使用 LOC 指标和功能点指标的度量结果。我们假定案例 1 使用汇编语言编写程序而案例 2 使用 C++ 语言编写相同程序。

案例 1: 基本汇编语言编写的应用软件 假定汇编语言编写的程序有 10 000 行代码,而各种书面文档(功能说明书、用户手册等)共有 100 页。假设编码和测试工作需要 10 个月,编写书面文档工作花费 5 个月时间。总项目工作需要 15 个月时间,因此该项目的生产率为每月 666 行代码。假设每个人月的成本是 10 000 美元,则项目总成本为 150 000 美元。用平均每行源代码成本来表示就是,每行源代码 15 美元。

案例 2: 用 C++ 编写的相同应用软件 假定 C++ 版本的相同应用软件仅需要 1000 行代码。由于使用面向对象(OO)语言开发,设计文档可能页数更少但用户手册与上述例子规模相同,所以假设书面文档总共 75 页。假设编码和测试需要 1 个月,而制作书面文档需

要4个月。那么,现在项目的总工作仅需要5个月,但用LOC指标表示,该项目的生产率却下降到了每个月只有200 LOC。如果每个人月的成本是10 000美元,则该软件的总成本为50 000美元,仅为汇编语言版应用成本的三分之一。C++版软件的成本比汇编语言版软件的成本便宜100 000美元,由此可以很清楚地看出,C++版具有更好的经济性。但是平均每行源代码的成本却已经跃升至50美元。

即使只度量编程工作,使用代码行度量指标,我们仍然无法看出高级编程语言的价值:即使C++版应用只需要1个月而基本汇编版应用需要10个月,汇编语言和C++语言的编码效率仍然相同,均为每个月1000行代码。

由于就功能而言,汇编语言版和C++版的功能完全相同,假设这两个版本应用软件规模均为50个功能点。当使用每个人月的功能点数来表示生产率时,汇编语言版本的生产率为每个人月3.33个功能点,而C++版本的生产率为每个人月10个功能点。让我们看看成本,汇编语言版应用软件的成本为每个功能点3000美元,而C++版应用的成本为每个功能点1000美元。因此,功能点度量指标清晰地符合了标准经济学的假设,该假设将生产率定义为:单位劳动或成本所产出的商品或服务。

另一方面,代码行指标并不符合标准经济学的假设,而实际上完全相反。代码行度量指标扭曲真正经济图景如此之多,在涉及一个以上编程语言的经济性研究中使用该度量指标可能会被归类为失职(professional malpractice)。

使用代码行的规模估算方法耗时情况 除非被估算规模的应用软件将要替代现存遗留应用,否则在真正编写完代码之前,这种规模估算方法纯属猜测。如果存在代码基准数据或来自类似项目的历史代码规模数据,假设新应用软件使用的编程语言与之前应用软件的编程语言也相同,这种规模估算方法可以较早完成。但是,如果没有任何历史数据或者旧应用软件的编程语言与新应用软件的编程语言不同,使用代码行指标的规模估算将无法精确完成,而且直到编写完代码之前都无法完成,而写完代码再估算则为时已晚。当新应用或者旧应用(或者两者都)使用多种编程语言时,代码计数将变得超级复杂且异常困难。

代码行规模估算法的使用情况 截至2009年,至少有300万个遗留应用软件仍在使用,而另有150万个应用软件正在开发中。然而在这总数450万个软件中,据笔者估计,超过400万个软件使用了多种编程语言或者使用的编程语言没有任何有效的代码计数规则。而在剩下总数大约50万个主要使用单一编程语言且该语言具有有效代码计数规则的软件中,使用了不少于500种编程语言。基本上,代码行规模估算法既不准确又充满危险,除非使用单一编程语言,如汇编、C语言、C语言的派生语言、COBOL语言、Fortran、Java以及其他近百种编程语言。

当今世界,尽管该方法有很多缺陷和问题,使用LOC指标的规模估算方法仍然经常被使用。国防和军事软件就是LOC指标最常见的使用者。LOC指标也仍然被系统软件和嵌入式软件广泛使用。老式的瀑布开发方法经常使用LOC规模估算方法,而现代团队软件过程(TSP)开发方法则同样如此。

进度与费用 如果存在可用的自动代码计数工具,则使用LOC指标的规模估算方法非常快速而廉价。但是,如果应用软件使用超过2种编程语言,自动代码计数则几乎不可能。

如果应用软件使用了某些现代编程语言，则使用代码计数完全不可能，因为在某些编程语言中用于“编程”的按钮和下拉式菜单还没有任何计数规则。

警告与注意事项 主要事项是代码行指标对高级编程语言非常不利。另一个注意事项是，这种规模估算方法对需求、功能说明书和其他书面文档的规模估算无能为力。同样，物理代码行的计数结果与逻辑语句的计数结果可能有超过 500% 的差异。鉴于软件文献和已发表的生产率数据在到底是使用了逻辑语句还是物理代码行上模糊不清，这种规模估算方法可能会产生巨大的误差。

6.2.3 基于故事点数指标的规模估算法

敏捷软件开发方法的出现部分原因是由于软件项目对表 6-6 所示的传统软件成本驱动因素的一种反应。敏捷先驱们认为，过多的书面需求文档和功能说明书已成为软件项目的沉重负担，而其中许多书面文档实际上对创建可工作软件似乎没有什么价值。

敏捷方法尽力简化并尽可能少产生书面文档，尽力提升创建可工作代码的能力。敏捷开发的理念是，软件工程的目标是以更具成本效益的方式创建可工作的软件程序。实际上，敏捷开发方法的目标是将传统软件成本驱动因素转变为更具成本效益的顺序，如表 6-7 所示。

表 6-7 敏捷软件成本因素排名

1	编程
2	会议和沟通（客户、团队成员、管理层）
3	缺陷去除（审查、静态分析、测试、发现和修复缺陷）
4	项目管理
5	撰写书面文档（各种计划、架构、功能说明、用户手册）

作为简化应用软件书面可交付物工作的一部分，一种称为用户故事（User Story）的方法被软件项目

用于收集用户需求。这些用户故事包含了特定软件需求非常简洁的描述，只有一两句话组成，被写在 3 英寸 × 5 英寸大小见方的纸片上以确保紧凑性。

例如，软件成本估算工具的一个基本用户故事可能是：成本估算工具应当包含美元、欧元和日元之间的货币换算。

创建之后，用户故事会被赋予一个称为“故事点”的相对权重，该权重反映了本用户故事相对于同一软件中其他用户故事近似的实现难度和复杂度。刚刚所举的货币转换的例子相当简单易懂（除非要求该货币转换必须基于每天的货币汇率波动），因此可能会被赋予 1 个故事点的权重。货币换算是一种非常简单的数学计算，网上也有很多现成的资源可用，因此这不是一个难以实现的故事或功能。

当然，同一款成本估算软件还将执行比货币换算更加困难和更加复杂的其他功能。例如，一个更加难以实现的用户故事可能会是：成本估算工具将能展示不同 CMMI 等级在软件质量和生产力方面的影响。

这个用户故事实现起来要比货币换算的用户故事困难得多，因为 CMMI 等级的影响随正开发软件的规模和性质而变化。对于小型简单软件，CMMI 等级几乎没有任何影响，但对于大型复杂应用软件，较高 CMMI 等级的影响非常显著。显然，上述用户故事的故事点数要比货币转换的故事点数多得多，可能会被赋予 5 个故事点的权重，这意味着实现这个用户故事的难度比货币换算的例子至少困难 5 倍。

特定软件的故事点数权重分配是由开发人员和用户代表联合制定的。因此，对于特定应用软件，各个故事点级别之间可能有高度数学一致性，即级别 1、2、3 等等，可能会有类似级别的难度。

敏捷文献通常会强调“故事点”是规模的单位，而不是时间或工作量的单位。然而话虽如此，实际上故事点经常被用于估算团队开发速度，甚至估算各个 Sprint 和软件项目的整体开发进度。

然而，用户故事和故事点指标非常灵活，因此不能保证两个不同应用软件的敏捷团队在赋予故事点权重时会精确相同。

随着敏捷方法获得越来越多的支持和广泛使用，也许会创建出确定故事点权重的通用规则并在实践中使用，但到目前为止这还只是设想。

当前，开发故事点与其他度量指标（比如 IFPUG 功能点、COSMIC 功能点、用例点、代码行等）之间的数学转换规则在理论上是有可能的。但是，若要如此，需要制定出应用软件之间保持使用一致性故事点的指导方针。换句话说，无论在什么地方应用用户故事，诸如 1 个故事点、2 个故事点等等数量必须具有相同的价值。

从上述故事点的例子看，就工作量而言，用户故事和故事点数之间似乎没有任何严格的线性关系。一个可能有用的近似情况是，假设用户故事点数每增加 1 点，与之相对应的 IFPUG 功能点增加一倍。例如：

当然，这种方法只是一种假设，但开展相关实验并创建故事点和功能点之间可靠的转换表将非常有趣。

如果敏捷社区能够收集有关工作量、进度、缺陷及其他可交付物的有效历史数据并提交给基准研究组织比如 ISBSG，这将对软件工程大有裨益。大量历史数据对故事点在估算方面的推广使用将非常有利，并且还能够加快故事点规模估算方法在诸如 COCOMO、KnowledgePlan、Price-S、SEER、SLIM 等商业估算工具软件中的推广应用。

故事点数	IFPUG 功能点数
1	2
2	4
3	8
4	16
5	32

除了故事点指标，一些敏捷项目还使用了功能点度量指标。但是，到本书写作的 2009 年为止，还没有敏捷项目向 ISBSG 或其他公共基准研究组织提交正式的敏捷项目基准数据。某些私有基准研究组织已经分析了大量敏捷项目，但研究结果为那些组织所专有，并未公开。

结果是，截至 2009 年，没有任何能够表明当前敏捷项目生产力或敏捷软件质量水平的可靠定量数据。这可不是一个专业工程行业的标志，相比于其他更加成熟的工程领域，这恰恰是“软件工程”多么落后的象征。

故事点规模估算法耗时情况 尽管敏捷项目竭力在整个软件项目启动时就给出一个整体概况，用户故事还是不断出现在整个开发过程中的每一个 Sprint 中。因此，使用用户故事的目的主要是为了当前 Sprint，而不太关注未来的下游 Sprint。因此，虽然对当前 Sprint 很有帮助，用户故事很难用于尽早地估算整个软件的规模。

故事点指标的使用情况 敏捷开发是一个非常流行的软件开发方法，但它离成为行业唯一的软件开发方法还很遥远。据笔者估计，截至 2009 年，约有 150 万个新软件正在

开发中。其中可能有 200 000 个软件项目使用了敏捷方法及故事点度量指标。故事点度量指标主要用于规模在 250 个到 5000 个功能点之间的小型到中等规模 IT 应用软件,既很少用于规模超过 10 000 个功能点的大型软件,又很少用于嵌入式、系统及国防软件。

进度和成本 由于故事点数是由团队一致非正式地分配给用户故事的,这种形式的规模估算方法快速而廉价。也有可能综合使用故事卡片和功能点。用户故事可以用作功能点分析的基础,但敏捷项目往往很少使用功能点指标。还有可能在敏捷项目中使用某些高速功能点方法,但截至本书写作之时,没有数据表明有敏捷项目正在这么做。

警告和注意事项 关于故事点的主要注意事项是对于特定应用软件其故事点往往是独一无二。因此,不太容易比较两个或多个使用故事点的不同敏捷项目之间的基准数据,因为无法保证不同应用软件的故事点使用相同的权重值。

另一个注意事项是,没办法将使用故事点度量指标的应用软件与使用功能点、用例点或任何其他软件度量指标进行规模估算的应用软件进行比较。故事点只能用于比较其他使用故事点指标的基准数据,即便最终数据并不准确。

第三个注意事项是软件行业还没有基于故事点的大规模基准数据集。由于某些原因,敏捷社区在基准研究和历史数据收集上非常宽松。这就是为什么很难确定敏捷方法比诸如 TSP、迭代开发甚至瀑布开发等软件开发方法具有更好还是更差的生产力和质量水平的原因。敏捷生产力和质量水平定量数据的缺乏是敏捷方法一个显而易见的弱点。

6.2.4 基于用例指标的规模估算法

用例 (Use Case) 自 1979 年就已存在,最初由 Ivar Jacobsen 提出,随后成为统一建模语言 (UML) 的一部分。用例还是 Rational 统一过程 (RUP) 不可分割的一部分 (Rational 公司已被 IBM 收购)。用例既有文字描述也有一些图形表示方法。除了 RUP,用例经常被应用于面向对象 (OO) 的应用软件,但有时也会被应用于非面向对象的应用软件。

用例从用户或者角色的角度描述了应用软件的功能。它有好几个详细程度等级,包括“简短”、“非正式”以及“完整正式”。其中“完整正式”用例最为详细,具有项目所需的足够详细信息,可以用于功能点分析及创建用例点 (Use-case point)。

用例中除了角色外还包括许多其他元素,比如前置条件、后置条件等。然而,这些元素都经过了严格定义,因而不同应用软件的用例在结构上非常一致。

用例和用户故事具有相似的视角,但用例更为正式且经常比用户故事大得多。由于用例存在的时间更长且研究用例的文献更多,不同应用软件的用例往往比用户故事更加一致。

一些批评意见认为用例无法处理非功能性需求比如安全和质量。但这样的批评也同样适用于其他任何设计方法。无论在何种情况下,将用例的使用扩展到质量、安全和其他非功能性设计问题都不太困难。

某种程度上,用例点的计算和逻辑与功能点指标在概念上颇为类似,只是具体细节有所不同。用例点的计算包括了技术和环境的复杂性因素。一旦计算出来,用例点数可以用来预测软件开发的工作量和成本。已有报道说平均每个用例的开发工作大概需要 20 个小时,但开发工作所包含的具体活动并不相同。

用例图及其支持文本可以用于功能点及用例点度量指标的计算。实际上,用例的严格性和一致性使其能够对用例点和功能点自动进行推导。

用例社区往往对功能点指标比较抵制,并声称用例和功能点关注的是应用软件的不同方面,而事实却并非完全如此。但是,既然两者都能产生每个点的工作时间信息,很显然二者之间具有的相似之处比用例社区承认的要更多。

假设每个月(work month)有22个工作日,每个工作日工作8小时,则一个月总共有176小时的工作时间。功能点生产率平均为每人月约10个功能点,即每个功能点需要17.6小时。

假定用例的生产率平均为每月8.8个用例,即等效于每个用例需要20小时。由此可见,用例点和IFPUG功能点产生的结果彼此相当接近。

其他作者及基准研究组织(比如David Consulting Group和ISBSG)已经发布了IFPUG功能点度量指标和用例点指标之间的转换比率数据。即使这些转换率数据与本章的上述数据不完全相同,它们也相当接近,而不同之处可能在于使用了不同的示例。

在用例点和COSMIC功能点、芬兰功能点或其他功能点变体之间可能也存在一个转换率,但笔者没有使用过任何其他的功能点变体,也没有搜索到它们相关的文献。

当然,使用IFPUG功能点和用例点的生产率可能变化很大,但整体上它们相差不大。

用例点规模估算法耗时情况 用例通常被用于定义软件需求及功能规范,因此当用例相当完整时可以计算出用例点数;也就是说,在接近需求阶段结束时才能计算功能点。不幸的是,往往在这个时间之前就需要正式的规模估算结果。

用例点的使用情况 RUP是一种非常流行的开发方法,但它离成为唯一的软件开发方法还差得远。据笔者估计,在2009年,约有150万个新应用软件正在开发中。其中,约75 000个软件使用了RUP方法及用例点度量指标。可能另外90 000个项目使用了面向对象开发方法及用例,但不是RUP。用例点可用于小型和大型软件项目。但是,数量庞大的用例常常使得大型应用软件开发变得复杂而低效。

进度和成本 由于用例点的计算比功能点更为简单,用例点规模估算方法比功能点分析稍快一些。相对于功能点分析的每天400点,每天可计算大约750个用例点。即使如此,如果使用手工规模估算,计算用例点的成本也高达每点3美元。很显然,自动规模估算将非常便宜,也更迅速。理论上,自动用例点规模估算每天可计算超过5000个用例点。

警告与注意事项 用例点的主要注意事项是目前尚没有大量使用用例点的基准数据。换句话说,还不能将用例点与行业数据库如ISBSG等进行比较,因为目前行业基准分析的最主要度量指标是功能点。

另一个需要注意的是,没办法将使用用例点度量指标的应用软件与使用功能点、故事点、代码行或其他软件度量指标进行规模估算的应用软件进行比较。即使用例点的结果数据非常稀少,很难找到这样的基准数据,用例点也只能用于与其他使用用例点指标的基准数据进行比较。

第三个注意事项是使用用例的项目中也没有广泛收集诸如生产力和软件质量等补充项目数据。由于某些原因,面向对象和RUP社区均在基准数据和历史数据收集上比较宽松。

这就是为什么很难确定 RUP 方法或面向对象方法是否比其他软件开发方法具有更好还是更差的生产力和质量水平的原因。在生产力和质量水平方面, RUP 方法缺乏可与其他开发方法(比如敏捷和 TSP 方法)进行比较的定量数据,这是用例点方法一个非常明显的弱点。

6.2.5 基于 IFPUG 功能点分析的规模估算法

为寻找一种不会像旧的“代码行”度量指标那样扭曲软件生产力经济性的度量指标,根据 IBM 公司高管的命令,经过不断研究和反复实验,IBM 公司的 A.J. Albrecht 和他的同事开发了一种称为“功能点”的度量指标,该指标与代码的数量无关。

功能点度量指标在 1978 年的一次大会上发布开放给公众使用。1984 年,IBM 将制定功能点度量指标计数规则的职责转交给了一个称为国际功能点用户组(IFPUG)的非营利性组织。

自从 1978 年推出功能点度量指标之后,基于功能点指标的软件规模估算技术成为可能。功能点规模估算方法比基于代码行的规模估算方法更为可靠,因为功能点指标支持所有的软件项目可交付物:书面文档、源代码、测试用例以及错误和缺陷。因此,功能点规模估算使软件项目规模估算工作从一个具有高错误率、非常困难的工作变成具有一定可接受精度的项目任务。

尽管当前的功能点计数规则仍然比较复杂,但功能点分析的实质是由包含如下 5 个元素的加权公式推导出来的:

1. 输入
2. 输出
3. 逻辑文件
4. 查询
5. 接口

该公式同时还包括复杂度的调整措施。具体的功能点计数规则由国际功能点用户组(IFPUG)发布,不在本节的讨论范围之内。

通过审查软件的需求文档和规格说明书可以量化功能点计数项。注意,传统的规格说明书、用例及用户故事都可用于功能点分析。功能点分析的计数规则还包括复杂度调整措施。功能点计数的确切规则不在本书要涉及的内容范围,因此这里不予讨论。

现在,功能点是世界上使用最广泛的软件规模估算指标,通过使用功能点提取所有项目主要可交付物(如项目计划和手册等书面文档、源代码及测试用例等)的有用规模数据,数以千计的软件项目能够很好地得以度量。下面是来自上述三种可交付物规模估算的一些例子。表 6-8 说明了为各种软件创建的典型文档数量。

表 6-8 所示的只是项目书面工作和文档规模估算能力的一个小例子,其中文档规模估算能力已开始在商业工具中可用。实际上,截至 2009 年,超过 90 种文档可以使用功能点进行规模估算,包括翻译为日语、俄语、中文等其他国家语言的文档。

功能点不仅可以用于估算书面可交付物的规模,还可以用于估算源代码、测试用例以及软件错误或缺陷的规模。实际上,功能点指标是任何已知度量指标中可以估算的软件可

交付物范围最广的指标。

表 6-8 为软件项目每个功能点创建文档的页数

	系统软件	MIS 软件	军用软件	商业软件
用户需求	0.45	0.50	0.85	0.30
功能说明书	0.80	0.55	1.75	0.60
逻辑说明书	0.85	0.50	1.65	0.55
测试计划	0.25	0.10	0.55	0.25
用户教程文档	0.30	0.15	0.50	0.85
用户参考文档	0.45	0.20	0.85	0.90
总文档数	3.10	2.00	6.15	3.45

对于源代码数量的估算,在近 700 种编程语言及其派生语言上已有可用数据。一些商业软件估算工具已有内置功能用于处理同一应用软件中的多个编程语言。

既然应用软件的功能点总数至少在需求阶段结束时是已知的,在规格设计阶段中期就更为详细,那就完全有可能相当精确地为那些使用了功能点指标的应用软件预测其估算规模。功能点规模估算法目前已成为诸如 COCOMO II、KnowledgePlan、Price-S、SEER、SLIM 等很多商业软件估算工具的标准功能。

IFPUG 功能点度量指标的广泛使用使其成为软件基准研究的标准度量指标。截至 2009 年,基于功能点指标的基准数据已经远远超过所有其他度量指标数据之和。当前,ISBSG 基准数据库包括约 5000 个项目的数据,并仍以每年约 500 个项目的速度增长。

由诸如 QPMG、David Consulting Group、软件生产力研究所 (SPR)、Galorath Associates 及很多其他公司拥有的专有基准数据库总共有约 60 000 个使用功能点指标的软件项目,且仍以每年约 1000 个项目的总体速率在增长。而目前还没有任何其他度量指标具有高达 1000 个项目的基准数据。

在过去的一些年里,有些担忧认为软件应用还包括很多“非功能性需求”,比如性能、质量等。事实的确如此,但这些非功能性需求的重要性被严重夸大了。

考虑一个建造房屋的例子。房屋建造成本的主要因素是以平方英尺或平方米计算的房子大小。房屋面积(平方英尺数)、舒适性以及建筑材料的质量品级都是用户需求。但是在笔者所在的州(罗德岛州),由于当地的建筑法规所规定的非功能性需求,建筑成本也显著增加。建在湖边、河边或当地地下蓄水层附近的房屋必须使用高科技化粪池系统,其花费比标准的化粪池系统贵 30 000 美元之多。建在距离大西洋海岸一英里范围内的房屋必须使用飓风防护窗,其花费比标准窗户贵出 3 倍之多。

这些政府强制要求不是用户需求。但如果房子不建在那些地方就不会有这些要求,它们可以被当做次要成本因素对待。因此,诸如“每平方英尺成本”等的估算和度量来源于功能性用户需求和政府建筑法规强制房主遵守的非功能性需求的组合。

IFPUG 功能点规模估算的耗时情况 IFPUG 功能点来源于应用软件的需求和功能规格说明,因此需求一旦完成就可以对功能点进行量化。然而,软件项目通常在需求完成之前就需要获取第一次正式成本估算结果。

IFPUG 功能点的使用情况 虽然 IFPUG 方法是使用最为广泛的功能点分析方法,但当前还没有任何一个功能点方法被整个软件行业广泛使用。截至 2009 年,大约总共有 150 万个新应用软件处于开发之中,据笔者估计,当前仅有约 5000 个应用软件使用了 IFPUG 功能点指标。另有约 2500 个应用软件可能使用了功能点的变体方法、逆火(backfiring)分析以及其他功能点粗略估计方法。由于自身的局限性,IFPUG 功能点很少在规模超过 10 000 个功能点的大型应用中使用,而且也根本无法应用于规模少于 15 个功能点的小型软件更新。

进度和成本 功能点规模估算方法既不快速也不便宜。功能点分析相当缓慢而且非常昂贵,以至于规模在 10 000 个功能点以上的应用软件几乎从来就没有完整地使用功能点方法进行过分析。

正规功能点分析必须由经过认证的功能点分析人员执行以保证准确性(未经认证的功能点分析人员的计数结果极不准确)。正规功能点分析每天可以计算出约 400 到 600 个功能点。以每天平均 3000 美元的咨询费计算,其平均成本在每个功能点 5.00 ~ 7.50 美元之间。

假定每个功能点的平均成本为 6.00 美元,那么估算 10 000 个功能点规模的应用软件将花费 60 000 美元。这就解释了正规功能点分析为什么通常只在 1000 个功能点规模数量级的应用软件上使用。

本章稍后会讨论各种形式的高速功能点粗略估计方法。应当指出的是,当软件开发项目使用了诸如用例等正式规格说明方法时,实现自动功能点计算是有可能的。

警告和注意事项 关于功能点分析的主要注意事项是,该方法非常昂贵而且相当耗时。虽然少于 1000 个功能点的小型应用软件在几天之内就能完成规模估算,但规模大于 10 000 个功能点的大型应用软件要好几周才能完成。由于功能点分析活动的高昂成本以及分析工作可能耗时数月这一事实,没有哪个规模超过 100 000 个功能点的超大型应用软件曾经真正使用功能点进行过规模估算。

另一个需要注意的是,功能点计数规则有时会改变。当计数规则改变时,基于旧版本计数规则的历史数据可能有所变化,或者无法与新版本计数规则所计算的功能点数据相兼容。这就要求必须有一个从旧计数规则到新计数规则的转换规则。如果非功能性需求确实是与功能性需求分开计算的,那这样的规则变化将会造成历史基准数据显著的不连续。

再一个注意事项是功能点分析有下限限制。由于功能点计算调整因子的下限限制,规模少于 15 个功能点的较小变更不能使用功能点分析进行规模估算。单个来看,这些小变更微不足道,但在大公司里每年可能有数以千计的此类小变更,所以此类变更的总成本有可能超过数百万美元。

关于功能点分析的一个警告是,精确的功能点分析需要由经过认证的功能点计数人员来执行,这些计数人员已经成功通过了由 IFPUG 提供的认证考试。由于功能点计数规则太复杂,软件项目不应该使用未经认证的功能点计数人员。就像税务法规一样,功能点计数规则的变化相当频繁。

功能点分析相当精确、非常有用,但其速度缓慢、成本昂贵。为此,软件行业开发了许多高速功能点分析方法,稍后将会详细讨论这些方法。

6.2.6 使用功能点变体的规模估算法

IFPUG 功能点度量指标的成功带来了一个奇特的现象。功能点度量指标的创始人——A.J. Albrecht——是一个电子工程师，当初将“功能点”定位为一个通用度量指标，可用于信息技术（IT）项目、嵌入式软件、系统软件、军事软件以及游戏和娱乐软件等各类软件。然而，第一次公布使用功能点度量的结果时，所使用的软件恰好是 IT 软件比如会计和金融软件。

功能点度量指标首先使用于 IT 应用软件这一历史性偶然事件使很多研究者得出这样的结论：功能点度量指标只适用于 IT 软件。结果，为了能够在其他类型软件上使用功能点度量指标，各种各样的功能点派生方法如雨后天春笋般诞生了，其中很多派生是针对系统和嵌入式软件的。这些功能点变体包括但不限于：

1. COSMIC 功能点
2. 工程功能点
3. 3-D 功能点
4. 全功能点
5. 特性点 (Feature point)
6. 芬兰功能点
7. Mark II 功能点
8. 荷兰功能点
9. 面向对象功能点
10. Web 对象功能点

当 IFPUG 功能点最初应用于系统和嵌入式软件时，有人指出这些应用软件的生产率较低。这是因为系统和嵌入式软件往往比 IT 软件更复杂一些，实际上更加难以构建，因此其生产率比同等规模的 IT 软件低约 15%。

然而，鉴于某些嵌入式和系统软件比 IT 软件更加难以开发因而具有较低的生产率这一事实，软件行业又开发出了很多功能点变体，这些变体方法表面上增加了嵌入式和系统软件的规模，因此看起来似乎比使用 IFPUG 功能点进行度量时的生产率高约 15%。

正如早先提到过的，仔细想想，这里有一件很有意思的事情。IT 软件似乎比系统和嵌入式软件具有更高的生产率，原因之一是 IT 软件项目的历史数据遗失比系统和嵌入式软件的历史数据遗失要多得多。这是因为 IT 软件通常是由某个成本中心开发的，而系统和嵌入式软件则通常是某个利润中心开发的。这种历史数据的遗失本身就足以使 IT 项目的生产率看起来比同等规模的系统和嵌入式应用软件的生产率至少高 15%。这或许是一个巧合，即由诸如 COSMIC 等功能点变体方法所预测的系统与嵌入式软件规模的增加恰好与 IT 软件历史数据的遗失率几乎完全相同。

并非所有功能点变体的出现都是因为要让某种软件的估算规模变得膨胀这种欲望，但很多功能点变体确实来源于此。结果是，当前术语“功能点”极其含混，它包含了很多变体方法。尽管可能一些功能点变体度量的结果比较相似，但软件行业不太可能混合这些功

能点变体而形成单个统一基准数据集，因为这儿有点儿像混合使用“码”和“米”或者“法定英里”和“海里”^①。

所有的功能点变体方法均声称其对于某种类型软件比 IFPUG 功能点具有更高的准确性，但这恰恰意味着这些功能点变体对于系统和嵌入式软件以及某些其他类型的软件能够产生比 IFPUG 功能点更大的计数结果。客观地讲，这与“准确性”可不能混为一谈。

实际上，完全没有任何客观办法可以确定 IFPUG 功能点及其各种变体的准确性。确定经过认证的功能点计数人员和未经认证的功能点计数人员之间在功能点计数结果上的差异以及执行功能点计算的两组功能点计数人员之间在相同测试用例上计数结果的差异倒是有可能的。但这并不是真正的准确性：它只是人类个体智力差异的外在表现而已。

功能点变体如此之多，将任何一种功能点变体应用于严格的项目规模估算和项目规划工作都非常困难。如果你恰好使用了某个功能点变体方法，那么你很有必要向管理你所使用的特定计数规则的协会或者组织寻求相应功能点计数规则的专业指导。

作为一项政策，功能点变体的发明者应负责制定该功能点变体与其他各种功能点变体之间的转换规则，以及与 IFPUG 功能点这种最古老、最原始的功能点度量方法之间的转换规则。但是，除个别特例外，当前各个功能点变体之间实际上没有任何有效的转换规则。目前只在 IFPUG 与 COSMIC 以及其他几种功能点变体（如芬兰和荷兰功能点变体）之间有某些转换规则。

较老的“特性点”度量指标是由 A.J. Albrecht 和本书作者联合发明的，已经经过校准以使其产生的计数结果与 IFPUG 功能点在 90% 以上的情况下相匹配；而对于剩下 10% 的情况，特性点度量指标的计数规则使其比 IFPUG 功能点产生更多的特性点结果，但二者之间可以通过数学方法进行转换。

现代社会很多度量指标都有多种变体，比如度量距离的“法定英里”和“海里”、度量容量的英制加仑和美制加仑、度量温度的华氏和摄氏。不幸的是，软件行业创造了比任何其他“工程”行业要多得多的度量指标变体。这正是软件工程还不算是一个真正工程学科的又一个标志，因为这说明软件行业还不知道如何高度精确地度量自己行业的数据结果。

功能点变体规模估算的耗时情况 IFPUG 功能点及其各种变体比如 COSMIC 等均来自于需求和功能规格说明，一旦初始需求完成马上就能量化其功能点数据。但是，软件项目常常在需求完成之前就需要第一次正式成本估算结果。

功能点变体的使用情况 IFPUG 功能点、COSMIC 功能点、荷兰功能点和芬兰功能点这 4 个功能点类型已经获得了 ISO 组织的认证。由于 IFPUG 功能点历史更为悠久，因而拥有最多使用者。COSMIC 功能点、荷兰功能点和芬兰功能点方法当前可能有 200 ~ 1000 个不等的应用软件在使用。较老的 Mark II 功能点方法可能约有 2000 个项目在使用，主要在英国。其他功能点变体方法每个估计有 50 个应用软件在使用。

进度和成本 计算功能点时，IFPUG 功能点、COSMIC 功能点和大多数功能点变体方

① 1 法定英里 = 5280 英尺，1760 码，约合 1.609 公里。1 海里 = 约 6076 英尺，约 2025 码，约 1.15 英里，1.852 公里。——译者注

法需要的时间基本相同。这些规模估算方法既不快速也不经济。各种功能点变体的分析过程均缓慢而昂贵,规模超过10 000个功能点的大型应用软件几乎从来没有使用功能点方法仔细分析过。

所有功能点变体的正规功能点分析均要求由经过认证的功能点分析人员来执行以保证准确性(未经认证的功能点分析人员产生的最终结果非常不准确)。正规功能点分析每天可完成400~600个功能点不等的计算。以每天平均3000美元的咨询费计算,每个所计算功能点的平均成本在5.00~7.50美元之间。

假设主要功能点变体的平均成本为每个功能点6美元,则计算一个规模在10 000个功能点的应用软件需要花费60 000美元。这就解释了为什么正规功能点分析通常只在规模在1000个功能点数量级的应用软件上使用。

警告和注意事项 所有功能点变体的主要注意事项是功能点分析太过昂贵且相当耗时。尽管规模小于1000个功能点的小型软件可以在数天之内完成规模估算,但规模大于10 000个功能点的大型应用软件则需要数周时间。正是因为功能点分析工作成本昂贵且耗时数月才能完成,目前还没有哪个规模超过100 000个功能点的超大型系统软件曾经真正使用IFPUG或者任何一种变体方法(比如COSMIC变体)进行过规模估算。

另一个需要注意的是,功能点分析方法的调整因子有一个下限。由于该下限的局限,规模小于15个功能点的较小变更不能使用功能点分析进行规模估算。对于所有的功能点变体方法比如COSMIC功能点、芬兰功能点等均是如此。单独来看,每一个小变更也许微不足道,但对于每年拥有数以千计这种小变更的大型公司来说,其总成本可能高达数百万美元。

一个警告是,准确的功能点分析要求必须由经过认证的功能点分析人员来执行,这些人成功地通过了由该功能点相关管理组织提供的认证考试。软件项目不应该使用未经认证的功能点计数人员,因为功能点计数规则过于复杂,就像税务法规一样,这些计数规则的变化也相当频繁。

功能点分析相当精确、非常有用,但其速度缓慢而且成本昂贵。为此,软件行业开发了许多高速功能点分析方法,稍后将会详细讨论这些方法。

6.2.7 使用功能点粗略估计的高速规模估算方法

功能点度量指标出现后没几年就有人指出了正规功能点分析的不足,即功能点计数速度缓慢、分析工作成本昂贵。实际上,非常早期的支持功能点度量指标的那些商业软件成本估算工具,比如1985年时的SPQR/20软件,还支持一种基于粗略估计而不是精确计数的高速功能点分析方法。

术语“粗略估计”是指当使用正规功能点分析的时候,不需要使用或者理解决定功能点规模大小的每一个因子就可以确定功能点的数量。

功能点粗略估计的业务目标是,所需时间保持在经过认证的功能点计数人员计算全部功能点所需时间的15%以内,也就是说只需少于一天的工作量就可得到全部结果。实际上,某些功能点粗略估计方法仅需一两分钟即可完成。功能点粗略估算方法并不能作为正规功能点分析的完全替代来使用,它们只是在软件开发早期软件需求还没有完全完成时提

供软件规模的一个快速粗略估计。之所以如此，是因为大多数软件项目在软件需求完成之前就需要有初始软件成本的估算结果，而此时还不可能进行正规功能点分析。

截至 2009 年，有很多种功能点粗略估计方法，其中使用最频繁的包括以下几种：

1. 未做调整的功能点方法
2. 简化复杂度调整的功能点方法
3. 轻量级功能点方法
4. 遗留应用数据挖掘的功能点方法
5. 基于回答应用软件有关问题的功能点^①方法
6. 基于模式匹配的功能点方法（本部分稍后会详细讨论该方法）

这些方法的目标是对正规功能点分析每天只能发现约 400 个功能点这一平均计数速度进行改善。也就是说，“未做调整”的功能点方法似乎可以达到每天近 700 个功能点的计数速度，使用简化复杂度因子的功能点方法可以达到每天计数约 600 个功能点的速度，而轻量级功能点方法则可以达到每天 800 个功能点的速度。

轻量级功能点方法是由 David Consulting Group 的 David Herron 开发的，他是一位认证的功能点计数员。他的轻量级功能点方法基于简化的标准功能点计数规则，尤其简化了复杂度调整。

基于遗留应用数据挖掘的粗略规模估计方法在技术上非常有趣。它是由一家叫做“Relativity Technologies”的公司（现已成为 Micro Focus 公司的一部分）开发的。对于 COBOL 和其他选定的编程语言，Relativity 功能点工具提取隐藏在遗留应用源代码中的业务规则，并将之用作功能点分析的基础。

该技术是由经过认证的功能点分析人员相互协作共同开发的，其计算结果只与标准功能点分析方法的结果相差几个百分点。相比标准功能点分析每天只有 400 个功能点，这种方法的名义速度可以达到每分钟 1000 个功能点。对于遗留应用软件，这种方法对功能点改造及使用它们来量化软件维护和功能增强工作非常有价值。

还有几种粗略估算是基于对应用软件提问的。软件生产力研究所（SPR）和 Total Metrics 公司均有此类工具可用。SPR 粗略估计方法已内置于 KnowledgePlan 估算工具里面。Total Metrics 粗略估算方法被称为“功能点概览”（Function Point Outline），涉及应用软件的一些有趣的外部属性，比如需求或功能规格说明的规模大小等。

正如本章早些时候指出的，功能点方法长期以来一直被用于度量和预测软件需求及功能规格说明的规模。“功能点概览”方法只不过逆推了功能点计算的数学公式，使用已知的文档大小来预测功能点数量，实质上是另一种形式的逆火（backfiring）分析。当然，文档大小只是所问问题的其中之一，该方法的基本想法是基于很容易获取的各种信息来创建功能点的粗略估计。

相比于正规功能点分析每天 400 个功能点的计算速度，“功能点概览”工具和其他基于

① 这种方法中，要求回答有关应用软件的一些问题，根据这些问题的答案，粗略估计出应用软件的规模。——译者注

问答的功能点粗略估计方法的速度基本上在每天 4000 个功能点左右。

基于功能点粗略估计的规模估算方法耗时情况 基于对应用软件问答的规模估算方法可以比标准功能点分析更早地应用于软件项目。轻量级功能点方法与标准功能点分析方法使用的时机基本相同,即当需求已知时才能使用该方法。数据挖掘方法要求有现存的源代码,因此主要使用于遗留应用分析。然而,对应用软件进行问答的粗略估计方法在需求阶段的非常早期就可以使用,这将比标准功能点分析的执行可能提前好几个月。

功能点粗略估计方法的使用情况 各种功能点粗略估计方法的使用情况相差很大。Relativity 方法和 Total Metrics 方法在 2008 年才开发出来,所以其使用数量还在不断增长,当前每种方法大概有 250 个项目在使用。每个较老的粗略估计方法可能有 750 个之多的项目在使用。

进度和成本 粗略估算方法的主要目的就是要实现比 IFPUG 功能点、COSMIC 功能点或者任何其他标准功能点分析方法更加快速的功能点计数速度和更为低廉的计数成本。他们的操作速度是标准功能点分析的 2 倍到 20 倍不等。而平均功能点成本则从少于 1 美分到 3 美元不等,但均比标准功能点分析方法便宜。

警告与注意事项 关于功能点粗略估计方法的主要注意事项是精确度。Relativity 方法几乎与标准 IFPUG 功能点的精度完全一致。其他粗略估计方法仅需认证功能点计数人员手工计数所需时间的 15% 以内,当然,计数结果比正规功能点计数提早近 3 个月,成本可能只是正规功能点计数的十分之一,这些都是重大的商业优势。

6.2.8 基于逆火分析或 LOC 到功能点转换的规模估算方法

概念“逆火”(backfiring)只不过是逆推了从功能点数量预测源代码数量的数学公式。逆火技术或者从 LOC 数据到等效数量的功能点的直接转换是由 Allan Albrecht 首先提出的,他还是功能点度量指标的发明人。大约在 1975 年,作为功能点度量指标初始开发工作的一部分,相关人员在 IBM 公司内部第一次收集了逆火数据。

第一个支持逆火技术的商业软件估算工具是 1985 年出现的 SPQR/20,它支持近 30 种编程语言的双向规模估算。今天,逆火技术已成为很多商业软件估算工具(比如本书早先曾提到的那些估算工具等)的标准功能之一。

从 1985 年的 30 种编程语言起,如果所有的派生语言都计算在内,到 2009 年可用于估算规模或逆火分析的编程语言数量已经增长到了超过 450 种之多。当然,对于那些不存在计数规则的编程语言,不可能进行逆火分析。软件生产力研究所每年都会发布编程语言逻辑代码行与功能点数之间转换比率的年度数据,而当前的 2009 年版包含了将近 700 种编程语言及其派生语言。其他咨询机构如 Gartner 集团和 David Consulting Group 等也曾发布过类似的数据。

有非常多的编程语言可以用来举例说明上述情况,但篇幅所限,无法尽述。还需要注意的是,逆火分析的边际错误也相当大。即便如此,其结果仍非常令人感兴趣,目前仍被广泛使用。下面的例子来自笔者文章《编程语言与等级表》(Jones, 1996),该表被软件生产力研究所每年更新好多次。该数据表明了对于给定编程语言构成一个功能点所需要源代

码语句数的范围和平均数。源代码计数规则基于逻辑语句，在笔者所出版的书籍《Applied Software Measurement》(McGraw-Hill, 2008) 的附录部分已详细定义。表 6-9 举例说明了逻辑源代码语句到功能点的转换比率。而所有 2500 种左右编程语言的完整转换比率表本书将不会一一列举，感兴趣的读者可查阅有关资料。

表 6-9 所选编程语言从逻辑源代码语句到功能点的转换比率

语言	名义等级	每个功能点的源代码语句数		
		低	中	高
第一代语言	1.00	220	320	500
基本汇编	1.00	200	320	450
宏汇编	1.50	130	213	300
C	2.50	60	128	170
BASIC (解释)	2.50	70	128	165
第二代语言	3.00	55	107	165
FORTRAN	3.00	75	107	160
ALGOL	3.00	68	107	165
COBOL	3.00	65	107	150
CMS2	3.00	70	107	135
JOVIAL	3.00	70	107	165
PASCAL	3.50	50	91	125
第三代语言	4.00	45	80	125
PL/I	4.00	65	80	95
MODULA 2	4.00	70	80	90
ADA 83	4.50	60	71	80
LISP	5.00	25	64	80
FORTH	5.00	27	64	85
QUICK BASIC	5.50	38	58	90
C++	6.00	30	53	125
ADA 9X	6.50	28	49	110
数据库	8.00	25	40	75
Visual Basic (Windows)	10.00	20	32	37
APL (默认值)	10.00	10	32	45
SMALLTALK	15.00	15	21	40
生成器 (Generators)	20.00	10	16	20
屏幕画图语言	20.00	8	16	30
SQL	27.00	7	12	15
电子表格	50.00	3	6	9

尽管通常逆火分析没有真正的功能点计数准确，但在某些情况下，逆火分析比功能点计算更为准确，例如，软件应用小于 15 个功能点的微小变更。对于小于 1 个功能点的代码变更，逆火分析是当前仅有的能够导出其功能点数量的两个方法之一。(第二个方法是模式

匹配,将在本节稍后详细讨论。)

虽然逆火分析已被广泛使用,也被众多商业软件成本估算工具支持,但该方法仍然像是一个“孤儿”,因为没有任何一个功能点用户组,如 IFPUG、COMIC 等曾经建立过专门委员会来评估逆火分析或者制定权威的逆火分析数据表。

逆火分析的一个潜在用途是,将用故事点或者用例度量的历史数据转换为功能点形式的数。这只需要推导出源代码逻辑语句数量,然后使用已发布的逆火分析转换比率即可得出功能点数量。

就如在编程语言的编译器中所做的,在各种代码分析器比如代码复杂度分析工具或者静态分析工具中添加逆火分析算法也算小事一桩。

虽然各个功能点协会忽视了逆火分析,但很多基准研究组织比如软件生产力研究所(SPR)、David Consulting 集团、QPMG、Gartner 集团等确实发布过逆火分析转换率数据表。

在各个公司的不同数据表中很多编程语言具有相同的级别,但如果使用不同公司的转换数据表,其他编程语言每个功能点包含源代码语句的数量则相差很大。这是一个非常尴尬的问题,如果不同咨询组织之间能够相互合作以尝试解决该问题的话,将对软件行业非常有益,但这种合作很可能不会发生。

令人颇感意外的是,截至 2009 年,所有已公布的逆火分析数据都与标准 IFPUG 功能点指标有关。制订诸如 COSMIC 功能点、故事点、用例点或任何其他度量指标的逆火分析规则非常容易,但不知为何,这个事情似乎从来没有人做过。

逆火分析功能点规模估算的耗时情况 鉴于逆火分析是基于应用软件源代码的,因此其主要用途是估算遗留应用软件的规模,用功能点来表示历史维护数据。逆火分析的第二个用途是将基于代码行(LOC)指标的项目历史数据转换为功能点数据,这样就可以将这些数据与行业基准数据(比如由 ISBSG 所维护的基准数据)进行对比。

逆火分析功能点的使用情况 逆火分析方法是作为创建功能点度量指标的一个副产品而由 A.J. Albrecht 创建的。因而业界自 1975 年以来就一直在持续使用逆火分析。由于速度快、简单易用,使用逆火分析进行规模估算的应用软件比其他任何度量指标都要多得多,笔者估计使用了逆火分析进行规模估算的应用软件多达 100 000 个。

进度和成本 如果源代码规模已知,用逆火分析进行规模估算既快速又便宜。如果使用自动代码计数,转换为功能点形式数据的速率可超过每分钟 10 000 行代码。相比于正规手工功能点分析每个功能点约 6 美元的成本,逆火分析的成本降到了每个功能点不足 1 美分。逆火分析还不需要经过认证的功能点计数人员来执行。当然,其准确性不是非常高。

警告与注意事项 逆火分析的主要注意事项是它不是非常准确。由于编程风格的差异,个体程序员之间在实现相同功能所需要代码行数量上的差异可达 6 倍之多。因此逆火分析的结果也相差悬殊。从数以百计使用相同编程语言(比如 COBOL)开发的应用软件使用逆火分析的结果看,比较合理的结果是程序和数据分区中平均约 106 个逻辑代码语句产生一个准确的功能点。但对于那些很少有应用程序使用的编程语言,则该值的变化范围非常之大。

第二个警告是,当前软件行业还没有代码行计算的标准方法。初始的逆火分析方法是基于逻辑语句数量的,如果用物理代码行来做逆火分析,产生的结果与相同例子使用逻辑

语句的逆火分析结果可能相差超过 5 倍之多。

还有一个注意事项是，对于代码中使用了 2 种或多种编程语言的应用软件，逆火分析会变得异常复杂。虽然存在处理任意数量编程语言逆火转换的自动化工具，但要想让这些工具很好地工作，必须知道每种语言在代码中的确切比例。

最后一个警告是，根据来源不同，已公布的表示代码行与功能点之间相互转换比率的规则也有所不同。由 David Consulting 集团、Gartner 集团、质量和生产力管理集团（QPMG）以及软件生产力研究所（SPR）发布的规则在相同编程语言上的转换比率确实不同。既然既没有哪个功能点管理协会（比如 IFPUG）也没有哪个大学曾经研究过逆火分析，那么也就不存在验证逆火分析假设条件的任何全面权威来源。

尽管其准确性令人质疑，逆火分析仍然颇为流行而被广泛使用。其受欢迎的原因正是由于正规功能点分析高昂的分析成本和漫长的分析周期。

6.2.9 基于模式匹配的规模估算方法

本节所有其他规模估算方法均为公开方法，任何人可以在任何需要的时候使用它们。但基于模式匹配的规模估算方法已提交了专利申请，因此这个方法可能还不能普遍使用。

模式匹配方法最初不是作为规模估算方法而出现的。它的开发首先是为了给基准研究提供一个明确无误的方法来识别和分类应用软件。在使用此分类方法度量了好几百款应用软件之后，有人指出在这种分类方法下相同模式的应用软件具有相同的规模。

模式匹配方法建立在这样一个基本事实之上，即已经存在了成千上万的遗留应用软件，且大量遗留应用软件的规模数据是已知的。可以通过包括现存应用软件的性质、范围、类别和类型等信息的分类方法，创建一个可用于新应用软件规模估算的模式。

使模式匹配方法真正有效工作的是一个包含了软件应用关键属性元素的分类方法。该分类方法由以下 7 个主要属性元素组成：（1）属性；（2）范围；（3）分类；（4）类型；（5）问题复杂度；（6）代码复杂度；以及（7）数据复杂度。每一个属性元素使用一个数字值作为标识。

在将一个软件项目与其他项目进行对比时，最重要的是要精确地知道所要比较的应用软件是什么类型。实际上这可没有看起来那么容易。软件行业本来就缺乏能够清晰而无歧义地用于识别和分类项目的软件项目标准分类方法，更别说在这个发明中所使用的分类方法了。

笔者曾提出过一个用以无歧义地分类软件项目的多部分分类方法。该分类方法版权归笔者所有，其详细内容已在笔者之前的几本书中做了详细解释，这些书包括《软件项目估计》（《Estimating Software Costs》，McGraw-Hill，2007）和《Applied Software Measurement》（McGraw-Hill，2008）。下面将再次解释该分类方法。

当该分类方法用于基准研究时，4 个额外因子来自公共来源，也是该分类的一部分：

这些代码来自国际电话区号、ISO 代码以及美国商务部的北美行业分类代码。这 4 个

国家代码	=	1	(美国)
地区代码	=	06	(加利福尼亚)
城市代码	=	408	(圣何塞)
NAIC 行业代码	=	1569	(通信)

代码不影响应用软件的规模, 但为软件项目基准研究和国际经济研究提供了颇具价值的信息。这是因为, 软件成本因国家不同、地理区域不同以及行业不同而差别巨大。历史数据要想真正有意义, 就非常有必要记录影响软件成本的所有因素。

用于估算应用软件规模的分类方法包括以下部分因子:

项目属性: _____

1. 新程序开发
2. 功能增强 (为现存软件添加新功能)
3. 软件维护 (修复现存软件的缺陷)
4. 转换或移植 (迁移到新的平台)
5. 再工程 (重新实现遗留应用)
6. 软件包修改 (修正购买的软件)

项目范围: _____

1. 算法
2. 子例程 (Subroutine)
3. 模块
4. 可重用模块
5. 一次性原型
6. 进化 (Evolutionary) 原型
7. 子程序 (Subprogram)
8. 独立程序
9. 系统的一个组件
10. 系统的一个发布版本 (初始版本以外的版本)
11. 新的部门级系统 (初始版本)
12. 新的公司级系统 (初始版本)
13. 新的企业级系统 (初始版本)
14. 新的国家级系统 (初始版本)
15. 新的全球级系统 (初始版本)

项目分类: _____

1. 个人程序, 仅私人使用
2. 个人程序, 其他人也可使用
3. 学术程序, 在学术环境中开发的
4. 内部程序, 只在单一地点使用
5. 内部程序, 可在多个地点使用
6. 内部程序, 在公司内网内使用
7. 内部程序, 由外部承包商开发
8. 内部程序, 通过分时共享使用其功能
9. 内部程序, 使用军用规格标准

10. 外部程序, 将发布给公众免费使用
11. 外部程序, 将放在 Internet 上进行共享
12. 外部程序, 租赁给用户
13. 外部程序, 与硬件进行捆绑
14. 外部程序, 分拆商业销售
15. 外部程序, 根据商业合同而开发的
16. 外部程序, 根据政府合同而开发的
17. 外部程序, 根据军事合同而开发的

项目类型: _____

1. 非过程式应用软件 (生成式 (generated)、查询、电子表格)
2. 批处理应用软件
3. Web 应用软件
4. 交互式应用软件
5. 交互式图形用户界面 (GUI) 应用程序
6. 批处理数据库应用程序
7. 交互式数据库应用程序
8. 客户端 / 服务器 (C/S) 应用程序
9. 计算机游戏
10. 科学或数学程序
11. 专家系统
12. 系统或支持程序, 包括 “中间件”
13. 面向服务架构 (SOA)
14. 通信或电信程序
15. 过程控制程序
16. 可信任系统 (Trusted system)
17. 嵌入式或实时程序
18. 图形、动画或图像处理程序
19. 多媒体程序
20. 机器人或机械自动化程序
21. 人工智能程序
22. 神经网络程序
23. 混合项目 (多种类型)

问题复杂度: _____

1. 没有计算或仅有简单算法
2. 大多数是简单算法和简单计算
3. 大多数是简单算法加上一些中等复杂度的算法
4. 均为简单和中等复杂度的算法和计算

5. 中等复杂度的算法和计算
6. 几个难以理解的算法再加上中等和简单的算法
7. 难以理解的算法比中等和简单算法多
8. 绝大多数是难以理解的和复杂算法
9. 难以理解的算法和某些极其复杂的算法
10. 所有算法和计算都极其复杂

代码复杂度：_____

1. 大多数“编程”工作是通过按钮或下拉式控件完成的
2. 简单非过程式代码（生成式、数据库、电子表格）
3. 简单及中等复杂度的非过程式代码
4. 由程序框架及可重用模块构建
5. 具有小模块和简单路径的中等结构化程序
6. 良好结构化的程序，但包括一些复杂路径或模块
7. 一些复杂模块、路径及不同程序段之间的链接
8. 中等以上的复杂度、路径及不同程序段之间的链接
9. 大多数路径和模块都是庞大而复杂的
10. 结构极其复杂，带有困难的代码段之间的链接和大型模块

数据复杂度：_____

1. 没有应用程序必需的永久性数据或文件
2. 只有1个必需的简单文件，和很少的数据交互
3. 一两个文件、简单数据及很小的复杂度
4. 几个数据元素，但有简单的数据关系
5. 多个文件，正常复杂度的数据交互
6. 多个文件，具有一些复杂数据元素和数据交互
7. 多个文件，复杂的数据元素和数据交互
8. 多个文件，复杂数据元素及数据交互占大多数
9. 多个文件、复杂数据元素、众多数据交互
10. 大量复杂文件、数据元素和复杂的数据交互

正如在软件项目度量和规模估算中最常见的那样，用户将为该分类方法的各个因子提供一系列的整数，例如：

项目属性	1	项目类型	15	数据复杂度	6
项目范围	8	问题复杂度	5	代码复杂度	2
项目分类	11				

虽然项目的属性、范围、分类、类型因子使用了整数，但三个复杂度因子则可以使用最多精确到小数点后两位的实数。模式匹配算法会在不同的初始整数之间进行插值以获得最合适的数值。这样，可允许数值如下所示：

项目属性	1	项目类型	15	数据复杂度	6.50
项目范围	8	问题复杂度	5.25	代码复杂度	2.45
项目分类	11				

该分类方法中各个因子数值的组合就构成了一个独特的“模式”，该模式使项目度量和规模估算变得更加容易。基于模式匹配规模估算方法的基本依据为如下两点：

1. 对大量应用软件的观测证明，在分类上具有相同模式的应用软件，其以功能点表示的规模大小也接近完全相同。

2. 上述分类方法 7 个因子的影响不尽相同。模式匹配的第二个关键是推导出每一个因子在确定应用软件规模上的相对权重。

要使用模式匹配规模估算方法，需要给每一个参数设置一个数学权重。具体权重值已在该方法的专利申请中做了详细定义，因涉及版权这里就不再赘述。然而，模式匹配规模估算方法的出发点是“范围”参数所涵盖的应用软件平均规模。表 6-10 说明了数学调整之前的平均值。

表 6-10 用于模式匹配规模估算的初始起始规模值

应用软件范围参数			应用软件范围参数		
值	定义	规模 (功能点)	值	定义	规模 (功能点)
1	算法	1	9	系统的一个组件	2500
2	子例程	5	10	系统的各个版本	5000
3	模块	10	11	新的部门级软件系统	10 000
4	可重用模块	20	12	新的公司级软件系统	50 000
5	一次性原型	50	13	新的企业级软件系统	100 000
6	进化原型	100	14	新的国家级软件系统	250 000
7	子程序	500	15	新的全球级软件系统	500 000
8	独立程序	1000			

如表 6-10 所示，软件应用的初始起始规模值依赖于用户提供给“范围”参数的数值。每一个答案都赋予了一个用 IFPUG 功能点表示的初始起始规模值。这些规模值是通过检查已用标准 IFPUG 功能点分析进行了规模估算的软件应用而预先确定的。初始规模值代表了那些已经使用功能点度量指标度量过的应用程序或者子组件的模式。

“范围”参数本身只提供了一个粗略的初始值。之后，必要时可根据分类、类型、问题复杂度、代码复杂度和数据复杂度等其他参数对该值进行调整。这些调整方法是该模式匹配规模估算方法专利申请的一部分。

软件行业不时会演变出许多新形式的软件。当有新形式软件出现时，可对上述分类方法加以扩展以包含这些新形式软件。

在软件项目的应用软件需求分析开始之前就可使用该分类。由于该分类包含了未来应用软件非常早期就已了解的信息，有可能在需求阶段完成之前几个月甚至在需求阶段开始之前就在项目中使用该分类信息。

也有可能将该分类信息用于那些已存在多年的遗留应用软件。知道这些遗留应用的功能点总数信息常常很有用，但由于遗留应用的需求文档和功能规格说明通常很少更新，有时可能根本就不知所踪，所以正常的功能点计数方法未必可行。

该分类信息还可用于商业软件，实际上可用于任何形式的软件，包括机密的军用软件，只要具有该软件足够的公开或保密信息用以给上述分类表中的各个参数赋予正确数值即可。

最初提出该分类方法时，使用的是 IFPUG 功能点以及逻辑源代码语句来度量产生的规模数据。但是，该分类方法也可以使用 COSMIC 功能点、用例点或者故事点等度量指标来估算软件的规模。要在该分类方法中使用其他度量指标，需要分析一下相关的历史数据。

基于模式匹配的规模估算方法可用于任何规模的应用软件,其范围包括从只有1个功能点一小部分的微小更新到可能多达300 000个功能点的大规模国防软件。表6-11说明了模式匹配规模估算方法在150个应用软件的抽样结果。使用此方法,每个应用软件都能在十分钟之内完成规模估算。

表6-11 模式匹配规模估算法在150个软件的抽样结果

注意1: 假设使用IFPUG版本4.2规则					
注意2: 代码数是逻辑语句数,不是物理行数					
应用软件	功能点规模 (IFPUG 4.2)	语言等级	源代码总数	每功能点代码行数	
1 星球大战导弹防御系统	352 330	3.50	3 2212 992	91	
2 Oracle	310 346	4.00	2 4827 712	80	
3 WWMCCS ^①	307 328	3.50	2 8098 560	91	
4 美国航空交通管制系统	306 324	1.50	6 5349 222	213	
5 以色列防空系统	300 655	4.00	2 4052 367	80	
6 SAP	296 764	4.00	2 3741 088	80	
7 NSA Echelon ^②	293 388	4.50	2 0863 147	71	
8 朝鲜边境防御系统	273 961	3.50	2 5047 859	91	
9 伊朗防空系统	260 100	3.50	2 3780 557	91	
10 宙斯盾驱逐舰作战系统	253 088	4.00	2 0247 020	80	
11 Microsoft Vista	157 658	5.00	1 0090 080	64	
12 Microsoft XP	126 788	5.00	8114 400	64	
13 IBM MVS	104 738	3.00	1 1172 000	107	
14 微软Office专业版	93 498	5.00	5983 891	64	
15 航空订票系统	38 392	2.00	6 142 689	160	
16 NSA 密码解密系统	35 897	3.00	3 829 056	107	
17 FBI Carnivore ^③	31 111	3.00	3 318 515	107	
18 人脑/计算机接口	25 327	6.00	1 350 757	53	
19 FBI 指纹分析系统	25 075	3.00	2 674 637	107	
20 NASA 航天飞机	23 153	3.50	2 116 878	91	
21 VA 病人监护仪	23 109	1.50	4 929 910	213	
22 F115 航空电子设备包	22 481	3.50	2 055 438	91	
23 律商联讯法律分析系统	22 434	3.50	2 051 113	91	
24 俄罗斯气象卫星	22 278	3.50	2 036 869	91	
25 数据仓库	21 895	6.50	1 077 896	49	
26 动画电影图形系统	21 813	8.00	872 533	40	
27 NASA 哈勃控制系统	21 632	3.50	1 977 754	91	

① World Wide Military Command and Control System, 全球军事指挥与控制系统。——译者注

② 美国两个强大的情报收集系统之一,受美国国家安全局(NSA)管理。——译者注

③ 美国联邦调查局(FBI)的电子邮件监视工具,用于实时监听因特网上传输的电子邮件。——译者注

(续)

	应用软件	功能点规模 (IFPUG 4.2)	语言等级	源代码总数	每功能点代码行数
28	Skype	21 202	6.00	1 130 759	53
29	舰炮控制系统	21 199	3.50	1 938 227	91
30	自然语言翻译	20 350	4.50	1 447 135	71
31	美国运通计费系统	20 141	4.50	1 432 238	71
32	M1 艾布拉姆斯主战坦克	19 569	3.50	1 789 133	91
33	波音 747 飞机航空电子设备包	19 446	3.50	1 777 951	91
34	NASA 火星探测器	19 394	3.50	1 773 158	91
35	Travelocity 在线旅行服务	19 383	8.00	775 306	40
36	苹果 iPhone	19 366	12.00	516 432	27
37	核反应堆控制系统	19 084	2.50	2 442 747	128
38	美国国税局所得税分析	19 013	4.50	1 352 068	71
39	巡洋舰导航系统	18 896	4.50	1 343 713	71
40	核磁共振医学成像系统	18 785	4.50	1 335 837	71
41	谷歌搜索引擎	18 640	5.00	1 192 958	64
42	亚马逊网站	18 080	12.00	482 126	27
43	订单录入系统	18 052	3.50	1 650 505	91
44	苹果 Leopard	17 884	12.00	476 898	27
45	Linux 操作系统	17 505	8.00	700 205	40
46	石油精炼过程控制	17 471	3.50	1597 378	91
47	企业成本核算	17 378	3.50	1588 804	91
48	联邦快递货运控制	17 378	6.00	926 802	53
49	战斧巡航导弹	17 311	3.50	1 582 694	91
50	化工炼油过程控制	17 203	3.00	1 834 936	107
51	ITT System 12 电信系统	17 002	3.50	1 554 497	91
52	Ask 搜索引擎	16 895	6.00	901 060	53
53	丹佛机场行李处理系统	16 661	4.00	1 332 869	80
54	ADP 工资单应用程序	16 390	3.50	1 498 554	91
55	库存管理系统	16 239	3.50	1 484 683	91
56	eBay 交易控制系统	16 233	7.00	742 072	46
57	爱国者导弹控制系统	15 392	3.50	1 407 279	91
58	Second Life 网站	14 956	12.00	398 828	27
59	IBM IMS 数据库	14 912	1.50	3 181 283	213
60	美国在线 (AOL)	14 761	5.00	944 713	64
61	丰田机器人制造系统	14 019	6.50	690 152	49
62	全州儿童支持系统	13 823	6.00	737 226	53
63	Vonage 公司 VOIP 系统	13 811	6.50	679 939	49
64	Quicken 2006 版	11 339	6.00	604 761	53
65	ITMPI 网站	11 033	14.00	252 191	23

(续)

	应用软件	功能点规模 (IFPUG 4.2)	语言等级	源代码总数	每功能点代码行数
66	机动车辆登记系统	10 927	3.50	999 065	91
67	保险理赔系统	10 491	4.50	745 995	71
68	SAS 统计软件包	10 380	6.50	511 017	49
69	Oracle CRM 功能包	6386	4.00	510 878	80
70	DNA 分析	6213	9.00	220 918	36
71	企业级 JavaBeans	5877	6.00	313 434	53
72	软件革新工具套件	5170	6.00	275 750	53
73	专利数据挖掘	4751	6.00	253 400	53
74	EZ Pass 车辆控制系统	4571	4.50	325 065	71
75	美国专利申请系统	4429	3.50	404 914	91
76	中国潜艇声呐系统	4017	3.50	367 224	91
77	微软 Excel 2007	3969	5.00	254 006	64
78	公民在线银行	3917	6.00	208 927	53
79	MapQuest 网上地图服务	3793	8.00	151 709	40
80	银行 ATM 机控制系统	3625	6.50	178 484	49
81	NVIDIA 显卡	3573	2.00	571 637	160
82	激光近视手术(波导)	3505	3.00	373 832	107
83	Sun D-Trace 工具	3309	6.00	176 501	53
84	微软 Office Outlook	3200	5.00	204 792	64
85	微软 Office Word 2007	2987	5.00	191 152	64
86	Artemis Views 软件	2507	4.50	178 250	71
87	ChessMaster 2007 游戏	2227	6.50	109 647	49
88	Adobe Illustrator	2151	4.50	152 942	71
89	SpySweeper 反间谍软件	2108	3.50	192 757	91
90	诺顿杀毒软件	2068	6.00	110 300	53
91	微软 Project 2007	1963	5.00	125 631	64
92	微软 Visual Basic	1900	5.00	121 631	64
93	Windows Mobile	1858	5.00	118 900	64
94	SPR KnowledgePlan	1785	4.50	126 963	71
95	全能打印机	1780	2.50	227 893	128
96	AutoCAD	1768	4.00	141 405	80
97	软件代码重构工具	1658	4.00	132 670	80
98	Intel 数学函数库	1627	9.00	57 842	36
99	索尼 PlayStation 游戏控制	1622	6.00	86 502	53
100	PBX [⊖] 交换系统	1592	3.50	145 517	91
101	SPR 检查点	1579	3.50	144 403	91

⊖ PBX, Private Branch (telephone) eXchange, 专用分组交换机。——译者注

(续)

应用软件	功能点规模 (IFPUG 4.2)	语言等级	源代码总数	每功能点代码行数
102 微软 Links 高尔夫球游戏	1564	6.00	83 393	53
103 GPS 导航系统	1518	8.00	60 730	40
104 摩托罗拉手机	1507	6.00	80 347	53
105 地震分析系统	1492	3.50	136 438	91
106 PRICE-S 软件	1486	4.50	105 642	71
107 响尾蛇导弹控制系统	1450	3.50	132 564	91
108 苹果 iPod	1408	10.00	45 054	32
109 物业税评估软件	1379	4.50	98 037	71
110 SLIM 软件	1355	4.50	96 342	71
111 微软 DOS 软件	1344	1.50	286 709	213
112 Mozilla Firefox 浏览器	1340	6.00	71 463	53
113 CAI APO (初始评估)	1332	8.00	53 288	40
114 Palm OS	1310	3.50	119 772	91
115 谷歌 Gmail 系统	1306	8.00	52 232	40
116 数字照相机控制软件	1285	5.00	82 243	64
117 IRA ^① 账户管理	1281	4.50	91 096	71
118 消费者信用报告软件	1267	6.00	67 595	53
119 激光打印机驱动程序	1248	2.50	159 695	128
120 软件复杂度分析器	1202	4.50	85 505	71
121 Java 编译器	1185	6.00	63 186	53
122 COCOMO II 软件	1178	4.50	83 776	71
123 智能炸弹瞄准系统	1154	5.00	73 864	64
124 维基百科站点	1142	12.00	30 448	27
125 音乐合成器软件	1134	4.00	90 736	80
126 配置控制软件	1093	4.50	77 705	71
127 丰田的普锐斯发动机	1092	3.50	99 867	91
128 人工耳蜗 (内部)	1041	3.50	95 146	91
129 任天堂 Game Boy DS 游戏机	1002	6.00	53 455	53
130 卡西欧原子手表	993	5.00	63 551	64
131 足球杯淘汰赛管理系统	992	6.00	52 904	53
132 COCOMO I 软件	883	4.50	62 794	71
133 APAR 分析与路由	866	3.50	79 197	91
134 计算机 BIOS	857	1.00	274 243	320
135 汽车发动机燃油喷射	842	2.00	134 661	160
136 防抱死制动控制系统	826	2.00	132 144	160
137 Quick Sizer 商业版	794	6.00	42 326	53

① IRA, Individual Retirement Account, 美国个人退休账户管理系统。——译者注

(续)

	应用软件	功能点规模 (IFPUG 4.2)	语言等级	源代码总数	每功能点代码行数
138	CAI APO (修订评估)	761	8.00	30 450	40
139	罗技无线鼠标	736	6.00	39 267	53
140	功能点工具	714	4.50	50 800	71
141	SPR SPQR/20	699	4.50	49 735	71
142	即时通软件	687	5.00	43 944	64
143	高尔夫障碍分析仪	662	8.00	26 470	40
144	拒绝服务病毒	138	2.50	17 612	128
145	Quick Sizer 原型	30	20.00	480	16
146	“我爱你”电脑蠕虫病毒	22	2.50	2838	128
147	按键记录病毒	15	2.50	1886	128
148	MYDOOM 电脑病毒	8	2.50	1045	128
149	APAR 错误报告	3.85	3.50	352	91
150	屏幕格式调整软件	0.87	4.50	62	71
	平均	33 269	4.95	2 152 766	65

由于模式匹配规模估算方法仍是实验性的且经常会被校准,表 6-11 中所示信息也是暂时的,可能经常会有变化,因而表中数据不应该用于任何严肃的商业目的。

注意,“语言等级”一列指的是 20 世纪 70 年代 IBM 开发的一个数学规则。“等级”的初始定义是完成高级编程语言中一个语句的相同功能所需基本汇编语言的语句数量。使用上述定义,COBOL 是一个“3 级”编程语言,因为需要 3 个基本汇编语句才能实现 1 个 COBOL 语句的相同功能。同样,Smalltalk 语言是一个“18 级”编程语言,而 Java 是一个“6 级”编程语言。

大约在 1975 年,当 IBM 开发出功能点度量指标时,同时也对当时存在的编程语言等级规则做了扩展,使其包含了平均每个功能点的逻辑源代码语句数目。

使用模式匹配无论是进行逆火分析还是预测源代码的规模,编程语言等级都是一个必需的参数。幸运的是,大约 700 个现存编程语言及其派生语言的等级已有公开数据可用。

模式匹配规模估算法的耗时情况 用于模式匹配的分类数据是通用的,所以甚至在完全了解需求之前就可以使用该分类数据进行规模估算。实际上,模式匹配方法是软件开发中可以最早使用的估算规模方法;远远早于正常功能点分析、故事点、用例点或任何其他度量指标。它也是唯一一个可以在需求分析开始之前使用的规模估算方法,因此能够在为软件项目提供任何资金之前获得一个有用的规模粗略估计。

模式匹配的使用情况 由于模式匹配规模估算方法是由一个专利申请所提出的,而且还处于试验阶段,截至 2009 年,该方法的使用只限于约 250 个试用该方法的应用软件。

应当指出的是,因为模式匹配根据外部应用分类而非该应用软件的具体需求,故模式匹配方法可以用于估算任何其他方法无法估算的应用软件的规模。例如,也许可以用该方

法来估算由其他国家（比如伊朗或朝鲜）开发的、极其机密的军用软件，任何一个国家都不会故意提供这些信息。

进度和成本 模式匹配规模估算方法内置于一个原型规模估算工具中，可以以超过每分钟 300 000 个功能点的速率预测应用软件的规模。这使该方法成为有史以来软件行业开发的最快和最便宜的规模估算方法。该方法如此快速、如此容易执行，因而可以很容易地执行最佳情况、理想情况和最差情况等好几种假设情况的规模估算。

即使没有自动化原型工具，模式匹配方法也可以使用袖珍计算器甚至手工以每个应用软件约两分钟的速度进行规模估算。

警告与注意事项 模式匹配的主要注意事项是该方法目前仍然处于试验阶段且会被不断校准。因此，估算结果可能会意外地发生变化。

另一个警告是需要使用大量具有标准功能点计数结果的历史项目来检验模式匹配方法的准确性。

6.2.10 估算软件需求变更的规模

到目前为止，所有已讨论的规模估算方法所产生的规模估算结果都只在某单一时间点有效。通过对大量软件项目的观察表明，软件需求在设计和编码阶段以每月 1% 到超过 2% 不等的速率增长或者变化。

因此，如果需求阶段结束时的初始规模估算结果为 1000 个功能点，那么该规模总数可能会发生增长，在设计阶段增长 6% 即 60 个功能点，编码阶段增长 8% 即 80 个功能点。当该软件最终发布时，其初始 1000 个功能点的规模将最终增长到 1140 个功能点。

软件规模增长与项目时间进度有关联。实际上，规模在 10 000 个功能点规模数量级或者更高数量级的大型应用软件，其规模总数的增长可能高达 35%，有些项目甚至高达 50%。很明显，这么多的规模增长将会对项目进度和成本均产生显著影响。

一些软件成本估算工具比如 KnowledgePlan 包含了预测需求增长率的算法，且允许用户选择接受或者拒绝这种预测结果。用户也可以在这些工具中添加他们自己的需求增长预测方法。

软件需求变化主要包括以下两种类型：

需求蔓延 这些需求变更会导致功能点总数的增加，因而也需要编写更多的源代码。软件项目应该估算这种变更的规模大小。当然，如果数量增加显著，还应该将它们包含到修订后的成本和进度估算中。

需求波动 这些需求变更不增加软件功能点规模总数，但可能也需要为它们编写更多代码。需求波动的一个例子是“改变输入屏的格式或外观但不增加任何新的查询或者数据库元素”。住宅建造的一个类比是用适合同一窗口开口的防飓风窗户替换掉现有的普通窗户。这个变化不会导致住宅平面面积的增加，但会增加住宅建造的工作量和成本。

软件应用的规模从未保持稳定，在开发期间会持续变化，即便是发布之后仍然如此。因此，规模估算方法需要能够处理软件需求的变更和增长，这些需求变化也会导致程序源

代码数量的增长。

需求蔓延比需求增长本身对软件项目的影响更加显著。当需求发生变更时,往往都非常匆忙,因而这些变更后的需求比初始需求的潜在缺陷发生率更高,往往也更加难以发现和消除其中的缺陷。如果需求变更发生在项目后期,迫于项目进度或成本等压力,项目人员可能会略过正式审查,而测试也可能做得非常不彻底。

结果是,大型软件项目的需求蔓延往往成为交付缺陷的主要来源,其中的交付缺陷数量比初始需求多得多。对于规模在10 000个功能点数量级的大型系统,几乎50%的交付缺陷可归咎于开发期间的需求蔓延。

6.3 软件进度与问题跟踪

在很多软件诉讼案件中担任专家证人的工作中,笔者注意到一个长期存在的软件项目管理问题。很多失败项目以及出现严重进度延误或重大质量问题的项目在开发期间通过正常的进度报告并没有发现任何问题。

从各种证人证词和案件举证中可知,软件工程师和一线项目经理均知道他们的项目中存在问题,但是当首次发现那些问题时,项目经理并没有第一时间将这些问题写进汇报给客户和高级管理层的状态报告里。直到很晚之后,当更高层或客户真正认识到严重进度延误、质量问题或其他重大问题时,形势通常已无法扭转。

当问到为什么要隐瞒项目中存在的问题时,得到的答案大多数都是基层经理们不希望高级管理层看到项目的糟糕状况。当然,当问题最终浮出水面时,实际上基层经理们看起来更糟糕。

相比之下,那些成功的软件项目总是以更加理性的方式处理项目中存在的问题。他们能尽早发现问题,组建专门任务小组来解决这些问题,且通常在这些问题变得严重到无法解决之前就能控制住局面。敏捷方法一个有趣的特色是每天讨论项目中存在的问题。团队软件过程(TSP)同样如此。

软件项目的问题有点儿像严重的医疗问题。它们通常不会自己消失,为消除这些问题,需要许多专业的“治疗”方法。

软件项目一旦启动,通常没有固定、可靠的指导性方法来判断项目实际进展速度。长期以来,民用软件行业一直使用特定里程碑的方法来确定项目进度,比如设计完成或编码完成等。但是,这些里程碑也是出了名的不靠谱。

软件项目状态跟踪需要涉及以下两个彼此不相关的问题:(1)达到具体、确定的里程碑;(2)在明确的预算金额内消耗项目资源和资金。

由于软件项目里程碑和成本受需求变更和“范围蔓延”的影响,当这些变化影响了功能点总数时,对需求变更的规模增长进行度量就变得非常重要。然而,正如本章上一节提到的,某些称为“需求波动”的需求变更不影响功能点规模总数,而需求蔓延和需求波动往往又都随机出现。需求波动比需求蔓延更加难以度量,往往只能通过程序源代码语句数

量和功能点指标之间的“逆火分析”或者数学转换来进行度量。

截至 2009 年，已有许多可用的自动化工具帮助项目经理记录项目里程碑报告所需要的各种重要信息。这些工具能够记录项目进度、资源、规模变化以及各种项目问题。

对于一个具有 60 年悠久历史的行业，却没有一套通用的或普遍的项目里程碑来标示项目实质性进展情况，这多少有些令人吃惊！笔者根据评估和基准研究得出的成果，给出了如下一些颇具实用价值、非常有代表性的里程碑。

表 6-12 显示的是某些大型软件项目中有代表性的跟踪里程碑。需要注意的是，这些里程碑均有一个与每一个主要软件可交付物的构建工作相关联的明确、正式的里程碑评审。在任何已知的软件质量控制活动中，正式检查和审查具有最高的缺陷去除效率水平，是一流组织的典型特征。

表 6-12 大型软件项目中的代表性跟踪里程碑

1	需求文档完成	22	变更控制计划评审完成
2	需求文档评审完成	23	安全计划完成
3	初始成本估算完成	24	安全计划评审完成
4	初始成本估算评审完成	25	用户信息计划完成
5	开发计划完成	26	用户信息计划评审完成
6	开发计划评审完成	27	特定模块代码完成
7	成本跟踪系统初始化完成	28	特定模块代码审查完成
8	缺陷跟踪系统初始化完成	29	特定模块代码单元测试完成
9	软件原型完成	30	测试计划完成
10	软件原型评审完成	31	测试计划评审完成
11	基础系统的复杂度分析（对于功能增强项目）	32	特定测试阶段的测试用例完成
12	基础系统代码重组（对于功能增强项目）	33	特定测试阶段的测试用例审查完成
13	功能性规格说明完成	34	测试阶段完成
14	功能性规格说明评审完成	35	测试阶段评审完成
15	数据规格说明完成	36	特定构建（Build）集成完成
16	数据规格说明评审完成	37	特定构建集成评审完成
17	逻辑规格说明完成	38	用户信息完成
18	逻辑规格说明评审完成	39	用户信息评审完成
19	质量控制计划完成	40	质量保证成功退出完成
20	质量控制计划评审完成	41	Beta 测试客户软件交付完成
21	变更控制计划完成	42	正式客户软件交付完成

表 6-12 最重要的地方在于这里的每一个里程碑均基于一个评审、审查或者测试活动的完成。只是完成了一个文档或者写完了一段代码并不能被视为一个里程碑的完成，除非这些可交付物经过了评审、审查或者测试并获得通过。

在笔者担任专家证人的那些案件中，这些标准通常都不具备。里程碑的完成标准很不正规，主要是根据日历日期而不是重要可交付物等材料本身的任何验证事宜。

同时,在那些案件涉及的软件项目中,里程碑报告的格式和结构也非常不完善。正常情况下,每一个里程碑报告顶部的显著位置应该突出显示并优先讨论项目中存在的问题或者“红灯”事项,但那些诉讼案件中的里程碑报告却常常不是这样。

在查看口供和法院案卷文件的过程中笔者注意到,软件项目人员和很多经理均已洞悉项目中存在的问题,而正是这些问题随后导致了项目延误、成本超支、质量问题以及法律诉讼。在低级别上,这些问题常常会写到项目周报中或在每日团队会议中加以讨论。但在给客户和更高层的高级别里程碑与项目跟踪报告中,那些危险的问题要么被遗漏,要么闪烁其词。

提交给客户和更高层的月度跟踪报告的建议格式可能包括以下这些部分:

软件项目月度状态报告的建议格式:

1. 本月出现的新“红灯”问题
2. 上月“红灯”问题的状态
3. 持续超过一个月的“红灯”问题的讨论
4. 本月实际处理的变更请求与预测的变更请求之间的差异
5. 预测的下月变更请求
6. 本月变更请求的功能点规模
7. 预测的下月变更请求功能点规模
8. 不影响功能点规模总数的变更请求
9. 本月变更请求对项目进度的影响
10. 本月变更请求对项目成本的影响
11. 本月变更请求对项目质量的影响
12. 本月发现的缺陷与预测的缺陷之间的差异
13. 本月发现缺陷的严重性级别
14. 缺陷起源(需求、设计、代码等)
15. 下月缺陷预测
16. 本月实际成本花费与预测成本之间的差异
17. 下月成本预测
18. 本月可交付物的挣值(如果使用挣值分析法)
19. 本月实际完成的可交付物与预测的可交付物之间的差异
20. 下月可交付物预测

尽管上述建议格式有点儿类似于项目使用的挣值计算方法,但该报告格式明确地讨论了变更请求的影响,还用功能点度量指标表示了成本和质量数据。

一个有趣的问题是,应当以怎样的频率向客户或高级管理层报告项目里程碑进展情况。最常见的报告频率是每月一次,但是任何时候当怀疑某些会导致项目出现动荡的情况发生时都可以提交项目异常报告。例如,关键项目人员的严重疾病或者离职都可能严重影响项目里程碑的完成,而这些情况是完全无法提前预测的。

可能有人会认为对于那些总体上仅持续6个月或者更短时间的小项目来说月度报告相隔时间太久了。对于小项目,周报可能确实更为理想。但是,小项目通常不会陷入成本超支、进度延误等严重麻烦,而大项目几乎总是麻烦重重。本书主要关注大型项目相关的项目问题。在笔者担任专家证人的那些诉讼案件中,除了仅1个项目外,其他所有项目的规模都超过10 000个功能点。

在项目同时部署并使用软件规模估算工具、项目评估工具、项目规划工具以及项目方法管理工具,能够在软件开发过程中为项目管理人员提供一系列清晰明了的控制点,允许项目管理人员在这些点上多少能够判断项目的一些实际进展。例如,当前的软件规模估算技术既可以预测软件规格说明的规模大小,又可以预测所需的源代码数量。缺陷评估工具能够预测项目中可能会遇到或发现的缺陷数目。尽管这些里程碑还不算完美,但它们也比以前的方法要好很多。

项目经理有责任为项目建立里程碑,监督其完成情况,并如实汇报这些里程碑是成功完成还是遇到了麻烦。当遇到严重问题时,有必要在汇报里程碑已完成之前先纠正这些问题。

失败或出现延误的软件项目通常都缺乏认真仔细的里程碑跟踪。经常出现的情况是,当项目活动被报告为“完成”之时,实际上相关工作却仍处于持续进行之中。失败项目的里程碑往往是日历日期而非实际可交付物及其评审完成。

交付的文档或代码段不完整、包含错误以及不能支持下游的开发工作,这些都不是行业领先者使用项目里程碑的方式。

行业领先者里程碑跟踪的另一个方面是,当问题被报告或者项目延误发生时会有什么处理。行业领先者对那些已报告问题的反应往往强劲有力、雷厉风行;规划纠正措施,分配任务小组,并立刻开始纠正报告的问题。而另一方面,对于行业落后者,报告的问题可能会被忽略而很少采取任何纠正措施。

在超过数十个牵涉到项目失败或项目从未成功运作的法律案件中,每一个案件中的项目跟踪都不够完善。项目问题要么被忽略,要么置之不理,而不是积极加以处理和解决。

由于里程碑跟踪存在于整个软件开发过程中,它们是预防项目失败或进度延误的最后一道防线。项目团队应当正式地建立项目里程碑,并对项目里程碑的可交付物进行评审、审查以及测试等验证。项目里程碑不应该是项目可交付物被完成的日期。它们应该反映已完成的项目可交付物被正式审查、测试以及质量保证评审等质量手段验证并获得通过的日期。

为持续跟踪面向对象项目的状态,Shoulders公司开发出了一种有趣的项目跟踪方式。这种方法使用了软件对象和类的3-D模型,使用由销子连接起来的各种大小的聚苯乙烯塑料泡沫球以创造出一种动态状态。整个展示结构被放置在一个尽可能多的团队成员都能看到的地方。球的动态变化使其状态立刻就能被所有观众看到。对颜色进行了编码的彩带表明每个组件的状态,不同的颜色分别表示设计完成、编码完成、文档完成以及测试完成(金色)等,还有某些颜色的彩带表明该组件可能出现了问题或延误。这种方法所提供的项目整体状态可见性几乎是瞬时的。该公司已经使用一种3-D建模软件包自动化实现了同样的方法,但物理展示结构更容易为人们所看见,并已在实际项目中证实了其更为有用。

Shoulders 公司的方法将大量重要项目信息浓缩到一个单一的可视化展示板上,使非技术人员也能很容易地理解其含义。

上述项目状态展示方法与关注项目问题及可能延误的每日状态例会相结合,将对软件项目非常有用。当向更高级经理或者客户提交正式书面报告时,其中包含的数据应当加以量化。此外,可能会导致项目延误的问题或者重大质量问题应该放在报告的首要位置并加以讨论,因为这些内容比报告中任何其他议题都更加重要。

6.4 软件基准

2009年初正当笔者写作本书时,国际标准化组织(ISO)发布了一个有关性能基准的新草案标准以供业界审核。目前该草案标准仍未获得最终批准。该草案涉及一系列的概念和术语定义,在随后的附加标准中对其进行了详细阐述。更多信息请查阅 ISO 组织相关网站。

软件度量和指标数据的主要商业用途之一是用于基准研究,即将公司的业绩与同行业或相关行业的类似公司进行对比。(同种数据也可用做过程改进度量的“基线”。)

术语“基准”的历史比计算机和软件专业的历史要长得多。它起源于木工工匠在工作台上标示标准长度的标志,其很快就传播到其他领域。基准的另一个早期定义是在测量领域,它表示一个刻有特定点的精确经度、维度和海拔高度的金属圆盘。术语“基线”同样来自于测量领域,它的最初定义是一条高度精确测量的水平线,用于高度和距离的三角测量。

当它开始在计算机行业使用时,术语“基准”最初用于规定处理器速度、硬盘和磁带驱动器的速度、打印速度等各种不同性能标准。该定义目前仍在使用中,而且最近几年来实际上已经为诸如 CD-ROM 驱动器、多频同步显示器、图形加速器、固态闪存磁盘、高速调制解调器等各种新型设备创建了许多新的和专业化的基准。

作为衡量计算机与软件行业中各种组织相对绩效的术语,20世纪60年代,术语“基准”首先被应用于数据中心的绩效考评。那是一个计算机进入业务运营主流的时代,各种数据中心的数量激增,数据中心的规模和复杂性不断增长。目前这种用法在判断数据中心运营的相对效率方面仍然很常见。

基准数据有很多用途,目前有很多收集和分析基准数据的方法。最常见和最重要的基准数据收集方式有以下5种:

1. **企业内部收集基准数据只在内部使用** 这种方式由某个公司或政府组织自己内部雇员负责基准数据的收集,只在组织内部使用。在美国,笔者估计约有15000个软件项目使用这种方式收集项目数据,主要是在那些大型和成熟的企业里,如 AT&T、IBM、EDS、微软等。这种内部基准数据为所属公司或组织所有,其他公司或者组织很少能够获得此类基准数据。以这种方式收集的内部基准数据的准确性差别很大。对于某些成熟的公司如 IBM,其内部基准数据非常准确。而对于其他一些公司,其基准数据的准确性则值得商榷。

2. **外部咨询顾问收集内部基准数据只企业内部使用** 第二种方式是,由外部基准咨询顾问负责收集基准数据,只在公司或政府单位内部使用。由于基准咨询顾问相当多,笔

者估计约有 20 000 个软件项目使用这种方法收集项目数据。以这种方式收集的基准数据属相关公司或组织所专有而不对外公布,除非不公开数据出处地将结果用于统计研究。之所以使用外部基准咨询顾问,是因为基准研究在技术上比较复杂,而由专家来做这项工作则通常要比未经培训的项目经理和软件工程师好得多。此外,基准顾问经验丰富,这有助于在基准数据收集过程中避免任何遗漏,发现更多其他问题。

3. 为外部公共组织或 ISBSG 基准组织收集内部基准数据 在这种形式中,由公司自己的雇员收集基准数据,提交给外部非营利性的基准研究组织,比如国际软件基准组织 (ISBSG)。据笔者估计,美国可能有 3000 个这样的软件项目已向 ISBSG 提交了项目数据。此类数据是现成的,公司和个人可以商业购买。提交给 ISBSG 的基准数据也可以通过估算、各种开发方法的有效性以及类似主题上的专题论文和报告而获得。这些基准数据的问卷调查由 ISBSG 提供给她客户,同时还附有如何收集此类基准数据的方法说明。使用这种方法来采集基准数据比较廉价,但由于公司与公司之间的答案并不一致,因而不同公司的基准数据可能差别很大。

4. 咨询顾问为外部营利性组织收集专有基准数据 这种形式是由外部咨询顾问负责收集相关项目的内部基准数据并将收集到的数据提交给咨询顾问所属的外部营利性基准研究组织,这些基准研究组织包括 Gartner 集团、David Consulting Group、Galorath Associates、质量与生产力管理集团、软件生产力研究所 (SPR) 以及其他类似组织。这些基准数据是由咨询顾问通过现场访谈方法收集的。据笔者估计,可能有 60 000 个软件项目通过外部营利性咨询组织来收集相关基准数据。这种方式收集的数据归相关公司所有,除非用于不公开数据来源的统计研究。例如,本书和笔者之前的书《Applied Software Measurement》,均使用了由笔者及其同事根据合同收集到的企业基准数据。但是由于保密协议,相关客户或项目的名字则不会被提及。

5. 学术性基准研究 这种形式是由大学学生或老师为学术研究之用而收集基准数据。据笔者估计,可能有 2000 个项目的基准数据是通过这种方式收集的。这些学术性基准数据可用于博士论文或其他论文,也可用于其他各种大学研究项目。某些学术性基准数据可能会以杂志文章或者书籍的形式得以公布。这类数据偶尔也可能会在市面上出售。学术性基准数据通常是通过电子邮件发放调查问卷(并附带如何填写这些调查问卷的详细说明)而收集来的。

所有这些基准数据来源加起来,其总数才约 10 万个项目。考虑到当前至少存在 300 万个遗留应用软件以及另有 150 万个新项目可能还在开发中,那么所有软件基准数据来源的总和才仅约占全部软件项目的 2%。

让我们聚焦于那些公众通过非营利性或商业来源可以获取的基准数据,全美国总共也只有约 3000 个项目的基准数据可用,只占全部软件项目的 0.7% 左右。这对于美国所创建的大量各种类别、类型和规模的软件来说,远小于统计学意义上达到的有效的最小样本量。笔者建议,来自 ISBSG 等非营利性来源的公共基准数据应当扩大到全部 150 万新开发软件项目的至少 2%,即约 30 000 个新项目。向公众提供全部遗留应用软件的至少 1% 即另外 30 000 个项目的基准数据抽样也将使软件行业从中受益。

当前基准数据的一个重大问题是项目规模分布不均。现有软件基准数据的绝大部分来自规模在 250 到 2500 个功能点之间的软件项目。只有非常少数的基准数据来自规模超过 10 000 个功能点的软件项目,而这些软件也是最昂贵、麻烦最多的应用软件类型。几乎没有任何规模小于 15 个功能点的小型功能增强项目的基准数据可用,即使这类项目比所有其他规模项目的总和还要多。

现有基准数据的另一个问题是项目类型分布不均。迄今为止,IT 项目的基准数据占到了全部基准数据的 65% 左右。系统和嵌入式软件项目的基准数据占约 15%,商业软件占约 10%,军用软件占约 5%。(鉴于国防部和军方所拥有的软件比这个星球上的任何其他组织都要多,军事软件基准数据的缺失很可能是由于很多军用软件项目都是保密的。)剩下 5% 包括游戏、娱乐、iPhone 和 iPod 应用以及如工具软件等五花八门的各种应用软件。

6.4.1 软件基准的类别

软件行业里各种软件基准种类的数量之多着实令人吃惊,而作为一项商业努力,这些软件基准使用了不同的度量指标、不同的度量方法以及针对软件的其他某种东西。

软件基准主要是那些表明应用软件种类、开发阶段或者软件活动生产率等的各种量化数据集。某些基准还包括以缺陷和缺陷去除效率形式表示的应用软件质量数据。此外,某个应用软件的基准还应该收集有关编程语言、工具和该应用软件所使用的开发方法等信息。

除了上述软件基准之外,软件行业还进行软件过程评估。软件过程评估收集各种详细数据,包括软件最佳实践和诸如项目管理方法、质量控制方法、软件开发方法、软件维护方法等等具体主题的详细数据。软件工程研究所(SEI)所开发的过程评估方法,用于评估一个组织的“能力成熟度等级”,这可能是世界上最著名的过程评估方式,不过也存在其他的几种过程评估方法。

由于过程评估数据和软件基准数据之间有着很明显的协同作用,因而软件行业还存在一些同时收集过程评估数据和软件基准数据的混合方法。这些混合方法往往使用庞大而复杂的问卷调查,常常通过现场咨询和面对面访谈的方式来获取数据。不过,使用电子邮件或基于 Web 的问卷调查以及使用诸如 Skype 或某些其他方法与软件工程师或者项目经理进行沟通以获取数据而不需要实际出差到项目现场,也是有可能的。

截至 2009 年,本书中所包括的主要软件基准有如下几种形式:

1. 国际软件基准
2. 行业软件基准
3. 整体软件成本与资源基准
4. 企业软件项目组合基准
5. 项目级软件生产力与质量基准
6. 阶段级软件生产力与质量基准
7. 活动级软件生产力与质量基准
8. 软件外包与内部开发绩效基准
9. 软件维护与客户支持基准

10. 软件方法基准
11. 软件过程评估基准
12. 混合评估与基准研究
13. 挣值基准
14. 软件质量与测试覆盖率基准
15. 软件质量成本 (COQ) 基准
16. 六西格玛基准
17. ISO 质量标准基准
18. 软件安全基准
19. 软件从业人员与技能基准
20. 软件薪酬待遇基准
21. 软件人员流动率或人员流失基准
22. 软件性能基准
23. 软件数据中心基准
24. 软件客户满意度基准
25. 软件使用情况基准
26. 软件诉讼与失败基准
27. 奖励基准

从上面这个长长的软件相关基准列表可以看出, 软件基准这一主题要比我们想象的复杂得多。

国际软件基准 伴随着经济衰退和全球软件竞争, 能够对全球范围内的软件开发实践进行比较正变得越来越重要。国际软件基准是一个相当崭新的领域, 但随着 Michael Cusumano、Watts Humphries、Howard Rubin、Edward Yourdon 及本书作者等的大量精品书籍出版, 该领域已经积累了大量相关文献。ISBSG 基准数据的一个不足是其数据来源的国家信息被刻意隐藏了。考虑到持续不断的经济衰退, 人们需要重新考虑这一策略。

进行国际软件基准研究的时候, 需要考虑很多当地因素。例如, 在日本, 每周都有至少 12 小时的无偿加班, 而其他国家比如加拿大和德国则很少有任何加班。在日本每周平均工作约 44 小时, 而加拿大却只有 36 小时。休假天数也因国家而异, 公共假期的数量也是如此。例如, 法国和欧盟国家的休假天数超过美国的休假天数达两倍之多。

当然, 软件外包最重要的国际化话题就是薪酬福利水平和通货膨胀率。因此, 国际软件基准研究要比国内软件基准研究复杂得多。

行业软件基准 随着经济衰退的持续, 不同行业之间在成本和薪水福利上的严重失衡越来越多地引起了人们的关注。例如, 银行家和金融高官们的高额薪水和巨额奖金已经令世界商业界感到震惊。尽管因为金额较小还没有达到众所周知的地步, 财务软件管理人员和工程人员挣得同样比其他行业的同类人员要多。随着经济衰退的持续, 很多公司都面临着这样的艰难抉择: 是投入大量资金和精力来改进他们自己的软件开发实践, 还是将所有的软件业务都转交给可能已经相当成熟的软件外包公司。有鉴于此, 各行各业的行业进度、

工作量以及成本等的基准数据将变得越来越重要。

截至2009年,人们已经获得了足够多的行业数据可以说明金融、保险、健康医疗、几种制造业、国防、医药与商业软件厂商之间的有趣差别。

整体软件成本与资源基准 企业层面的成本与资源情况研究本质上类似于经典的数据中心基准研究,只是专注点转移到了软件开发组织。这些研究收集如下数据:人员和设备的年度支出、雇用的软件人员数量、所服务的客户数量、软件项目组合的规模以及其他与软件开发和维护有关的具体方面。然后,将收集的结果数据与行业标准或者来自相似规模公司、同一行业公司或者那些具有足够多共同点使得该对比非常有意义的公司等平均数据进行比较。这些高层次的基准数据经常是由“战略”咨询机构如麦肯锡、Gartner集团等收集并整理的。整体软件成本与资源基准不涉及单个项目,而是重点关注企业级或业务群级的费用模式。

在那些具有多个办公地点的超大型企业里,相似的基准数据有时也可用来比较公司内部的不同地点或者部门之间的状况。大型会计公司和很多管理咨询公司可以从事一般成本和资源基准研究工作。

企业软件项目组合基准 企业的一个项目组合可以达1000万个功能点之多的规模,其中包含超过5000个应用软件。这些应用软件可以包括IT项目、系统软件、嵌入式软件、商业软件、工具软件、被外包的应用软件以及开源应用。很少有哪个公司能够清晰无误地知道他们的项目组合里到底有多少个应用软件。考虑到整个项目组合也许就是这些公司所拥有的最有价值的资产,项目组合级基准数据的缺乏实在令人不安。

由于大型公司的项目组合规模极其庞大,而收集这些大型公司全部软件的海量基准数据所需成本又极其昂贵,因此目前项目组合的基准数据很少。

笔者曾参与过一个大型制造企业集团的项目组合基准研究,该项研究历时大约12个月,参与研究的10名咨询人员走访了该企业集团遍布至少24个国家的60家子公司。仅是这一个项目组合基准研究的数据收集就花费了超过2百万美元。不过,该项目组合本身的价值达到约150亿美元。对该公司来说,这是一笔非常重要的公司资产,因而非常值得花费大量资金和精力对其进行研究并理解其状况。

当然,对于那些项目组合主要集中在单一某个数据中心的小型公司,它们的这种基准研究可能只需几名咨询人员在个把月内就能完成。但不幸的是,大型公司通常在地理上十分分散,他们的项目投资组合也是高度分散在很多城市和国家。

项目级生产力与质量基准 项目级生产力与质量基准将关注点从整个组织层面下移到了项目层面,专门收集特定项目的项目数据。这些项目级基准研究不断积累项目工作量、项目进度、人员编制、成本以及质量等项目数据,而这些数据均取自负责基准数据收集的组织所开发和/或维护的软件项目。有时,抽样项目比例可高达100%,但大多数时候抽样项目数较为有限。例如,某些公司不考虑低于一定最小规模的项目,如50个功能点;或者排除那些处于开发之中、仅供公司内部使用而不会发布给外部客户的软件项目。

有时候,通过电子邮件或者直接发给参与者问卷或者调查工具进行项目级生产力和质量基准研究。项目级基准数据包括项目进度、以小时或月度量的工作量以及项目成本,

可能还会包含编程语言和开发方法等补充数据。虽然质量数据应该包含在项目级基准数据中,但实际上却很少有人这么做。

为了避免“风马牛不相及”的比较,进行项目级基准研究的公司通常会对项目数据进行分类划分,这样的话,系统软件、信息系统、军事软件、科学软件和其他类型的软件将可以与同类型的软件项目进行比较。还会将基准数据以应用软件的规模进行分类,确保不会将规模很小的软件项目与巨型软件项目进行比较。同样还要区分新开发项目、功能增强项目和软件维护项目。

尽管在项目层面上收集基准数据做起来相当容易,但是目前还没有任何简便的方法来验证基准数据的准确性,也没有办法可以确保不会遗漏大量的项目工作及相关项目成本数据。有鉴于此,项目级基准数据的准确性总是令人存疑。

阶段级生产力与质量基准 不幸的是,软件行业目前还基本上不太可能有办法验证项目级基准数据的准确性,因此那些数据往往并不可靠。若将基准研究的关注点下移到项目阶段层面上,则可以提供更加精细的数据粒度,因而也更具价值。截至2009年,关于术语“阶段”,软件行业还没有普遍认同的标准定义。但通常的阶段模式包括需求、设计、开发和测试这四个阶段。

当将基准研究作为软件过程改进活动的前奏时,经常还会用到类似术语——基线。这时候,基线反映的是当进行基准研究时已经存在的项目生产力、进度、人员编制和/或质量水平。然后,这些结果可用于将来以一定的时间周期度量项目进展情况或者项目改善情况。基准和基线收集相同的信息,因而也基本相同。项目级数据对于基线来说毫无用途,因而,阶段级项目数据是能够显示项目过程改进结果的最低粒度级别。

阶段级基准被ISBSG所采用,它还频繁地被使用在学术研究中。实际上,大量软件基准文献所涉及的往往都是阶段级基准数据。目前,现存可用的阶段级基准数据已经非常多,这些数据相当准确地为美国软件行业确定了项目数据平均值及偏差范围,以及为许多其他国家确定了软件项目的初步平均值。

活动级生产力与质量基准 不幸的是,仅收集项目数据的度量措施无法进行准确性验证,而阶段级数据也同样难以验证其准确性,因为很多项目活动(比如技术文档写作和项目管理活动)都是跨阶段的。

基于项目活动的基准比上文已经讨论过的项目级基准更为详细。活动级基准将关注点下移到为构建应用软件而必须执行的具体类型的项目工作层面。例如,自20世纪80年代以来笔者就一直使用25个项目活动来划分具体的活动级基准,这25个项目活动包括:需求活动、原型构建、软件架构、项目规划、初步设计、详细设计、设计评审、代码编写、可重用代码采购、软件包采购、代码审查、独立验证与确认、配置控制、模块集成、用户文档、单元测试、功能测试、集成测试、系统测试、现场测试、验收测试、独立测试、质量保证、软件安装和项目管理。

基于项目活动的基准研究比其他类型的基准研究更加难以执行,但其结果对项目过程改进、成本缩减、质量改进、进度加快及其他类型的改进计划都更为有用。活动级基准的很大优势在于它们揭示了其他粒度的基准研究很少能够提供的极其重要的信息种类。例如,

对于很多类型的软件项目，主要项目成本驱动因素都与书面文档（各种计划、功能说明书、用户手册）的制作以及质量控制（审查、静态分析、测试）等紧密相关。书面工作成本和缺陷去除成本经常比编码成本昂贵得多。上述发现对于规划项目改进计划和计算投资回报都有很大帮助。但是，要想清晰了解某个具体公司或企业的主要成本驱动因素，有必要首先着手仔细探讨活动级的基准研究。

尽管当前可能会使用 Skype 或者电话会议，但项目活动级基准研究通常通过现场访谈的方式来收集数据。典型地，这个级别的基准访谈需要约 2 个小时时间，需要项目经理参加访谈，可能还会需要至多 3 名项目团队成员也参加访谈。因此，总时间约为 8 个人工时，再加上收集基准数据本身所需要的咨询时间。如果咨询师还需要计算功能点数，那么他们需要花费更多额外时间。

软件外包与内部开发绩效基准 笔者受托开展软件生产力与质量基准研究的最常见原因之一是，很多公司都在考虑将它们软件开发业务的一部分或者全部工作外包出去。

通常，能够做出外包决策的都是公司里 CEO 或 CIO 级别的高层人员。低层管理人员因为可能会失去工作而惶恐不安，因此他们会委托专业人员进行生产力与质量研究，以将公司内部软件开发的业绩与行业数据以及美国国内及海外的主要软件外包公司的业绩进行对比。

直到最近一段时期，与中国、俄罗斯、印度以及其他的软件外包国家相比，就每月开发完成的功能点数而言，美国软件开发的业绩还算相当不错。但是，如果算开发成本，较低的海外劳动力成本给了离岸外包强有力的竞争优势。不过，过去数年内，海外国家的通货膨胀率比美国上升得要快一些，这使得二者的成本差别已经大大缩小。例如，由于与其他地方相比生活成本较低，IBM 最近决定在爱荷华州兴建一个大型外包中心。

持续不断的经济衰退导致美国软件专业人员供大于求，这也降低了美国软件行业的薪酬福利水平。结果是，许多国家之间的劳动力成本开始趋于平均。经济衰退也影响着其他国家，但是由于出差成本持续上升，开展海外业务正开始变得越来越难，至少是不那么方便了。

软件维护与客户支持基准 截至 2009 年，从事软件维护和功能增强工作的软件工程师比从事新软件开发的软件工程师数量多很多。然而，软件维护和功能增强工作的基准却常常很少有人做。之所以如此，原因众多。其中一个原因是，软件维护工作包括了不少于 23 种类型各异的遗留应用更新，其范围从轻微的代码更改到完全的软件革新均有涉及。另一个原因是，大量的软件维护工作涉及的变更规模小于 15 个功能点，而这样的规模小于正规功能点分析所要求的最低下限。尽管单独来看这些小变更可能会快速完成，所以比较廉价，但是由于大型公司里存在着成千上万的此类小变更，其累积成本总和每年可高达数百万美元。

一个颇具价值的软件维护工作关键度量指标是“维护任务量”，即一个工程师负责维护或运行的软件数量。其他的维护度量指标包括所支持的用户数、缺陷修复率、用每月功能点数或每个功能点工作小时数形式表示的正规生产率。同时，潜在缺陷和缺陷去除效率水平也很重要。

关于软件维护基准的一个强烈警告是：传统的“平均缺陷成本”度量指标具有严重缺陷，它对软件质量非常不利。平均缺陷成本度量指标使漏洞百出的软件也能达到平均成本最低。

尽早发现缺陷并予以修复似乎要比晚发现并修复的成本更为便宜,但考虑到与缺陷无关的固定管理费用而不是与缺陷有关的实际时间花费及具体活动,这一结论实属错误。

信息技术基础架构库 (ITIL) 中包含的有关服务和客户支持的新要求给了软件维护和客户支持基准一个新的推动力量。实际上, ITIL 基准应该成为软件基准的一个重要分支领域。

软件方法基准 当前, 软件行业存在很多种不同形式的软件开发方法, 比如敏捷开发、极限编程 (XP)、水晶开发、瀑布开发、Rational 统一过程 (RUP)、迭代开发、面向对象 (OO) 开发、快速应用开发 (RAD)、团队软件过程 (TSP), 等等。同时, 还存在大量的混合开发方法以及数以百计可能仅在某个公司内部使用的定制化或本地方法。

除了开发方法之外, 很多其他方法也会对软件开发的生产力、质量或者二者均产生影响。这些方法包括六西格玛、质量功能展开 (QFD)、联合应用设计 (JAD) 以及软件重用。

基准数据应当具有合适的粒度、足够的完整性以表明各种开发方法相关的生产力和质量水平。ISBSG 基准数据就非常完整, 足以表明上述情况。同样, 营利性基准研究组织比如 QPMG 和 SPR 等所收集的数据也能说明上述情况, 但其具有一定的逻辑问题。

这些逻辑问题包括: 某些比较流行的开发方法比如敏捷方法和 TSP 方法使用了非标准度量指标, 比如用户故事点、用例点、理想时间以及任务小时数。使用这些度量指标收集的基准数据与主流行业基准数据并不兼容, 因为所有的行业基准数据均基于功能点度量指标和标准工作时间。

另一个逻辑问题是, 使用某些较新开发方法的组织很少委托外部顾问或者使用 ISBSG 数据调查问卷进行基准研究。因此, 很多软件开发方法的有效性仍然含糊不清、难以确定。将非标准度量指标数据转换为功能点和标准工作时间在技术上没有障碍, 但是, 敏捷社区或者大多数使用非标准度量指标的其他开发方法还没有这么做的。

软件过程评估基准 软件过程评估方法自 20 世纪 70 年代就已在 IBM 等大公司中使用。大约在 1986 年, Watts Humphrey 离开 IBM 并为软件工程研究所 (SEI) 创建了软件评估方法之后, IBM 风格的软件评估方法就逐渐流行起来。巧合的是, 大约在 1984 年, 本书作者离开 IBM 并为软件生产力研究所 (SPR) 创建了自己的软件评估方法。

自从两本书出版之后, 软件过程评估突然受到了公众的极大关注。其中一本书是 Watts Humphrey 的《软件过程管理》(《Managing the Software Process》, Addison Wesley, 1989), 该书详细描述了软件工程研究所 (SEI) 所使用的软件过程评估方法。关于软件过程评估的第二本书是本书作者撰写的《Assessment and Control of Software Risks》(Prentice Hall, 1994), 这本书描述了软件生产力研究所 (SPR) 所使用的软件过程评估方法的评估结果。由于本书作者和 Watts Humphrey 在 IBM 工作时均参与过 IBM 的软件过程评估工作, 因而 SEI 和 SPR 的软件过程评估方法具有一些共同特征, 比如均极为强调软件质量。

SEI 和 SPR 软件过程评估方法在概念上均类似于身体检查。也就是说, 这两种方法都会试图找出公司构建和维护软件的过程中任何正确或者不正确的事情。希望不会有太多错误的东西, 但在开出任何真正有效的“治疗方案”之前, 有必要先弄清楚到底哪里出了问题。

巧合的是, SPR 方法和 SEI 方法均使用了一个“5-点”等级来评估软件过程性能。不幸的是, 这两种方法的软件过程性能等级方向完全相反。SPR 的软件过程性能等级基于里

氏等级,较大数字表明危险越来越严重。而 SEI 的软件过程性能等级使用数字“1”作为最原始评分,而随着逐渐增加到“5”级,软件过程也变得越来越严格。下表是 SEI 评分系统及其 5 个等级中每个等级已知企业数量的近似百分比。

由此可见,所有使用 SEI 评估方法的企业中约有 75% 的企业处于最低等级,即“初始级”。还需要注意的是,SEI 评分系统缺乏中间点或者平均值。

有关 SEI 评分系统的深入讨论不在本书讨论范围之内。SEI 评估方法的得分是根据对一套约 150 个是/否型问题的回答结果进行评估而得出来的。较高的 SEI 成熟度等级要求对特定问题模式回答“是”。

下面是 SPR 评分系统以及军用软件、系统软件和管理信息系统(MIS)软件等 3 个行业分组内企业评估结果的大概比例。

SEI 能力成熟度模型(CMM)评分系统

等级定义	百分比
1=初始级	75.0%
2=可重复级	15.0%
3=已定义级	7.0%
4=量化管理级	2.5%
5=优化级	0.5%

SPR 软件评估评分系统

定义	百分比(整体)	军用软件	系统软件	MIS 软件
1=优秀	2.0%	1.0%	3.0%	1.0%
2=良好	18.0%	13.0%	26.0%	12.0%
3=中等	56.0%	57.0%	50.0%	65.0%
4=不佳	20.0%	24.0%	20.0%	19.0%
5=糟糕	4.0%	5.0%	2.0%	3.0%

SPR 评分系统很容易描述和理解。它基于整套 SPR 评估问卷中大约 300 个 SPR 问题的平均答复进行评估而得出结果。

通过 SPR 评分的反演和数学压缩,在 SPR 评估等级和 SEI 等级之间有可能建立一个粗略的对等关系,如下表所示:

SPR 评分范围	等效 SEI 等级	近似百分比	SPR 评分范围	等效 SEI 等级	近似百分比
5.99 ~ 3.00	1=初始级	80.0%	1.01 ~ 2.00	4=量化管理级	3.0%
2.99 ~ 2.51	2=可重复级	10.0%	0.01 ~ 1.00	5=优化级	2.0%
2.01 ~ 2.50	3=已定义级	5.0%			

当然,上述 SPR 和 SEI 评估结果之间的转换并不完美,但它确实让使用某种评估方法的用户对“使用另一种评估技术进行评估其结果看起来可能怎样”有了一个大概的了解。

当前,软件行业还存在着很多其他形式的软件过程评估方法。例如,ISO 质量认证使用一种软件评估方法,而欧洲则使用 SPICE^①和 TickIT 方法。

尽管很多组织确实拥有自己的内部评估专家,但通常情况下,软件过程评估工作都是由外部咨询师来执行的。对于 SEI 风格的软件评估,很多咨询组织已获得授权可以开展评

① ISO 15504 标准,又称为 SPICE,全称为“软件过程改进与能力确定”(Software Process Improvement and Capability Determination),是一种与 CMM 非常类似的成熟度模型。——译者注

估研究工作并收集相应数据。

混合评估与基准研究 基准数据可以说明生产力和质量水平，但却无法解释是什么原因导致这样的结果。过程评估数据可以说明软件开发实践的成熟程度，或者相同开发实践的缺乏情况，但是软件评估常常并不收集量化数据。

很明显，评估数据和基准数据具有协同互补作用，而且都需要进行收集。笔者建议将过程评估数据和软件基准数据进行合并，这将对软件行业非常有益。实际上，笔者自己的基准研究中总是混合使用评估和基准方法，同时收集评估数据和基准数据。

混合基准的关键优势之一是定量数据能够展示较高 CMM 及 CMMI 等级的经济价值。而没有经验基准数据，提升 CMMI 等级从 1 级到 5 级的价值就仍然是未知数。但是与 CMMI 的 1 级和 2 级的生产力和质量水平相比，基准数据确实可以说明 CMMI 3 级、4 级和 5 级的生产力和质量水平要高得多。

软件行业从过程评估方法与软件基准数据收集方法的广泛整合中受益匪浅。混合方法的优势在于它最大程度地减少了项目经理或者技术人员接受访谈或者被要求提供相关信息所花费的时间，这能够防止评估与基准数据收集活动妨碍软件项目的实际日常工作。

在大公司或者政府团体中，需要进行整合以获得其整体软件状况的某些类型数据包括：

1. 关于团队规模的人口数据
2. 关于软件专家的人口数据
3. 关于集中办公与分布式团队的人口数据
4. 使用几种度量指标（功能点、用户故事点、LOC 等）的应用程序规模
5. 可重用代码和其他可交付物的数量
6. 开发期间的需求变更率
7. 关于项目管理方法的数据
8. 关于软件开发方法的数据
9. 关于软件维护方法的数据
10. 关于具体编程语言的数据
11. 关于所使用的具体工具套件的数据
12. 关于质量控制与测试方法的数据
13. 关于潜在缺陷和缺陷去除效率水平的数据
14. 关于软件安全控制方法的数据
15. 项目活动级的进度、工作量及成本数据

过程评估与基准数据收集活动的共同使用能够以相当具有成本效益及无干扰的方式收集上述所有类型的数据信息。

挣值基准 将项目实际累积工作量及成本与预测的里程碑和可交付物进行比较的挣值方法广泛应用于军事应用软件中；而事实上，挣值方法的确是军事软件开发合同的硬性要求。但是，在国防软件社区之外，一些外包合同也使用挣值计算方法，而某些公司内部应用软件偶尔也会使用挣值计算方法。

挣值方法通常会以一定的时间间隔计算项目挣值，通常是每月一次，用以指示项目实

际进展与费用情况。尽管当前已有许多软件工具可用于计算挣值，但挣值方法仍然有点儿过于专业，挣值计算也很复杂。

挣值方法本身并不是一种真正的软件基准，因为它的关注点非常狭窄，并不涉及诸如软件质量、需求变更等主题以及其他项目问题。但是，为挣值方法收集的项目数据对基准研究相当有用，也可以说明其与评估结果（比如软件能力成熟度模型集成（CMMI）的级别）之间的相互关系。

软件质量与测试覆盖率基准 在非营利性基准组织如国际软件基准组织（ISBSG）等提供的公共基准数据中，软件质量并没有很好地得以体现。实际上，包括微软等主要软件厂商在内，整个软件行业在软件质量方面做得都不是很好。

像 IBM 这样的公司能非常严肃地对待软件质量，他们认真地度量从软件需求阶段、开发阶段一直到发布并进入客户现场之后所发现的所有缺陷。目前软件行业使用两个非常重要的度量指标用于创建软件质量基准，这两个指标是：潜在缺陷和缺陷去除效率。术语“潜在缺陷”指在整个软件中可能发现的所有缺陷总数。术语“缺陷去除效率”指每次评审、审查、静态分析及各个测试活动所发现并修复的缺陷占全部软件缺陷的百分比。

此外，软件质量基准可能还包括如下几个主题：程序流图循环复杂度和基本复杂度、测试覆盖率（测试用例实际调用到的代码占全部代码的百分比）以及缺陷严重性等级。目前的行业基准数据中缺乏很多质量相关数据，比如测试用例本身的缺陷或者错误。

通常来讲，软件行业需要数量更多、质量更好的软件质量与测试覆盖率基准。但是目前包含诸如测试用例数目、测试的运行次数以及缺陷去除效率水平等信息的测试文献非常稀缺。

关于软件质量基准的一个强烈警告是，“平均缺陷成本”不是一个可以安全使用的度量指标，因为它对软件质量极为不利。笔者把该度量指标看作接近是失职。而质量经济学中一个更好的度量指标是“平均每个功能点的缺陷去除成本”。

软件质量成本（COQ）基准 不幸的是，如此重要的思想却有着“质量成本”这样如此不合适的名字。正如 ITT 的 Philip B. Crosby 所指出的那样，质量不仅是免费的，而且还具有经济价值。COQ 度量本应该用于命名像“缺陷成本”之类的东西。不管怎样，质量成本方法要比软件和计算机行业的历史更为悠久，它来自于约瑟夫·朱兰（Joseph Juran）、爱德华·戴明（W. Edwards Deming）、石川馨（Kaoru Ishikawa）、田口玄一（Genichi Taguchi）以及其他很多质量先驱人士。

传统的 COQ 成本组成要素包括预防、评估及失败成本。虽然这些成本要素在软件上也可行，但软件 COQ 经常具有更多的成本要素，比如缺陷预防、正式审查、静态分析、软件测试以及交付缺陷修复等。两者的基本想法是一样的，但使用了不同的命名以符合软件行业的实际情况。

很多公司在其应用软件和工程产品上均会进行质量成本基准研究。有关该主题，目前存在大量的文献资料，并已出版了几十本参考书籍。

六西格玛基准 “六西格玛”是一个数学表达式，它将缺陷数量限制为每 100 万个机会里不能超过 3.4 个缺陷。虽然这个如此严格的定量结果在软件上几乎是不可能的，但六西格

玛的理念可以轻而易举地应用于软件行业。

六西格玛方法采用了一套相当先进和复杂的度量指标来考察软件缺陷起源、缺陷发现方法、交付给客户的缺陷以及其他相关主题。同时，六西格玛方法还使用这些数据来改进缺陷预防和缺陷检测。

六西格玛方法存在很多种变形，其中当前最重要的变形是“精益六西格玛”，它尝试使用一种极度简化的方法对缺陷与软件质量进行数学分析。

就传统意义来讲，六西格玛方法不是一个真正的软件基准。就像通常所使用的，软件基准是有限时间段内所收集数据点的一个离散集合，比如收集某个电信公司在 2009 年所开发的 50 款应用软件的数据。

六西格玛方法在时间上并不固定，也不限制应用软件的数目。它是数据收集、分析和改进三者持续不断的循环，一旦启动，就会无中断地持续进行下去。

尽管六西格玛的理念非常强大，而且往往非常有效，但是当应用于软件时，在相关文献和实践数据方面就存在着极其显著的不足。截至 2009 年，没有足够的经验数据能够说明六西格玛在软件行业的推广应用提升了缺陷去除效率水平或者降低了潜在缺陷。

大约在 2009 年，美国软件行业的整体平均潜在缺陷为平均每个功能点约 5.00 个缺陷，而缺陷去除效率平均只有约 85%。二者组合的结果是，当软件最终被交付给用户时，平均每个功能点中仍然遗留了 0.75 个缺陷。

鉴于六西格玛度量指标的统计特性，有意思的是将所有使用精益六西格玛或者软件六西格玛的公司与美国平均数据进行比较将会怎样。如果这样比较，有人可能会希望六西格玛方法的潜在缺陷将会更低（假设每个功能点约 3.00 个缺陷），而缺陷去除效率则会更高（假设高于 95%）。但不幸的是，这种数据非常稀缺，软件行业还无法提供足够数量的此类数据以进行令人信服的统计研究。

碰巧的是，能够实现软件六西格玛的一种方法就是设法达到 99.999% 的缺陷去除效率，而实际上这在软件行业从未发生过。但是，将缺陷去除效率的实际水平与六西格玛的理论目标进行比较看来还算很有用处。

从历史的角度看，缺陷去除效率计算并非源自六西格玛领域，而是源自 20 世纪 70 年代早期的 IBM 公司。当时 IBM 公司需要将软件审查与其他形式的缺陷去除活动进行比较，这就提出了缺陷去除效率的计算问题。

ISO 质量标准基准 那些需要 ISO 9000 ~ 9004 质量标准认证或者其他较新相关 ISO 标准认证的组织，需要接受对他们的质量控制方法和过程——尤其是质量控制方法的相关文档——进行现场检查。这种认证是某种形式的基准研究，但实际上开展起来相当昂贵。但是，目前很少或根本没有任何经验数据可以说明 ISO 认证提高了哪怕是一丝一毫的软件质量。

换句话说，经过 ISO 认证的组织既没有在潜在缺陷方面也没有在缺陷去除效率水平方面比类似未经认证的组织似乎更好一些。实际上，已有传闻说，那些未经认证的公司的平均软件质量可能略高于那些经过认证的公司的软件质量。

软件安全基准 除了美国国土安全部、联邦调查局及最近美国国会的研究报告外，当

前,几乎完全没有任何企业级别的软件安全基准。随着经济衰退的延续及软件安全攻击事件不断增加,软件行业迫切需要建立软件安全基准以对软件安全方面的重要主题进行度量,这些主题包括:软件抵御攻击的能力、每家公司或每款应用软件遭受攻击的次数、安全漏洞预防的成本、从安全攻击和拒绝服务攻击中恢复的成本以及最有效安全防护方法的评估等。

软件杂志里确实包含了很多安全基准信息,比如易于使用的反病毒与反间谍应用软件、防火墙、检测到的病毒信息或者病毒穿越防护墙的情况等。然而,这些基准信息有点儿含糊不清、漫不经心。

到目前为止,可以确定的是还没有任何已知的关于软件安全主题的基准信息,比如针对微软 Vista、Oracle、SAP、Linux、Firefox 浏览器、IE 浏览器等的安全攻击次数方面的信息。如果软件领域能有这些安全主题的月度基准信息,将对软件安全防护非常有益。缺乏有效的软件安全基准是一个明显的信号,它说明软件行业在软件安全问题上还没有全面行动起来。

软件从业人员与技能基准 软件从业人员与技能储备基准是软件行业刚刚兴起的一个崭新主题。软件已经成为全球业务的主要因素之一。某些大型企业拥有超过 5000 名各类软件专业人员,相当数量的公司拥有的软件专业人员数量超过了 2500 人。除了上述数量庞大的软件工人,与软件相关的各类具体技能与职业种类的总数量已接近 90 个。

正如前几章中讨论过的,除了一般的软件工程人员之外,大型企业还拥有为数众多的不同类型专家,比如,质量保证专家、集成与测试专家、人为因素专家、软件性能专家、客户支持专家、计算机网络专家、数据库管理专家、技术通信专家、软件维护专家、项目估算专家、软件度量专家、功能点计数专家,等等。

当前有这么几个重要问题需要回答:对于某一个项目,到底需要多少名不同类型的专家;企业该如何招聘、培训专家以及在专家们的专业领域给予认证。另一个重要问题是如何以最优化的方式在整个软件组织结构中组织、管理和使用这些专家。在因持续不断的经济衰退而导致的企业业务规模不断缩减和业务流程再造(BPR)这样的大背景下,该领域的基准研究还需要收集不同行业不同规模的公司如何应对不断增长的专家需求方面的信息。

由于持续不断的经济衰退而越来越重要的一个新话题是在美国工作、持有临时工作相关签证的外国软件工作者的分布情况。这个话题最近受到新闻界的持续关注,因为有人指出,微软和 Intel 公司裁减美国软件工人的速度要比裁减国外软件工人的速度快得多。

软件薪酬待遇基准 薪酬基准已在非软件行业的薪酬研究中使用超过 25 年了。当软件行业蓬勃发展时,研究人员又很快将软件行业的薪酬数据添加到了此类部分公开或者不公开的基准中。

薪酬基准的使用方法是,许多软件公司向薪酬咨询公司提供他们付给使用标准工作职位描述所定义的各类软件工作者的薪酬福利水平数据。中立性的薪酬咨询公司分析这些数据,并向各个公司报告研究的结果。每一份报告会特将特定公司的薪酬数据与整个数据组的平均数据进行比较。在“部分公开”形式的薪酬研究报告中,会指出其他公司的名称,但是当然不会提及这些公司的实际薪酬数据。在“不公开”形式的薪酬研究报告中,参与该基准研究的公司数目是已知的,但不会指出具体的公司名字。以不公开或部分公开形式开展薪酬研究有其法律上的原因,因为这些研究会牵涉反垄断法规,也许会面临“相互串通垄

断控制行业薪酬待遇水平”的指控。

软件人员流动率与人员流失基准 在软件成为主要业务职能之前，该基准已在软件之外的行业里广泛使用。当发展到规模足够庞大以至于人员流失成为一个重要问题时，软件组织只不过是引入了该项基准研究。

人员流失及流动率基准研究通常是由人力资源组织负责实施的，而不是软件组织。这些基准通常都是经典的不公开或部分公开形式的基准。一般来讲，数十个甚至数以百计的软件公司向中立性的外部咨询团体报告他们的人员流动率数据，外部咨询团体随后返回每个公司的统计分析结果。在返回的报告中，会将该公司的流动率与所有提交数据进行比较，但不会提及其他公司的具体流动率情况。

在诸如 IBM、谷歌、微软、EDS 以及类似的其他大型公司里，同样也会开展内部人员流失率研究。笔者曾接触过很多非常重要的此类内部研究数据，其中最重要的结论之一是，个人绩效考核得分最高的软件工程师离开的人数最多。在很多离职谈话中提到的最常见离职原因是，优秀的技术人员不愿意再为不称职的经理工作了。

软件性能基准 软件运行速度即软件性能是最古老的软件基准之一，自 20 世纪 70 年代就已开始开展此类研究。鉴于需要考虑应用软件在各种不同情况下的吞吐量或者执行速度，这种基准是技术含量很高的软件基准。几乎每一本个人电脑杂志都开辟有诸如图形处理、操作系统加载时间以及其他性能问题等主题的基准研究专栏。

软件数据中心基准 这种形式的基准可能是计算机和软件行业里历史最悠久的软件基准，自 20 世纪 60 年代就已开始持续开展此类基准研究。数据中心基准需要收集诸如软硬件的可用性、应用软件的平均无故障时间以及缺陷修复时间间隔等主题的信息。新版信息技术基础架构库 (ITIL) 包含了很多需要审查的主题，可以将它们包含到服务协议里。

尽管数据中心基准与软件基准有所不同，但二者也有所重叠，因为数据中心性能不佳往往与所安装软件的糟糕质量水平有关联。

软件客户满意度基准 长期以来，像 IBM、惠普、Unisys、谷歌等计算机和软件厂商一直实施正式的客户满意度调查，而某些小型公司也开展此类调查。这些基准研究通常由市场部门负责具体执行，用于为他们的商业软件包提供改进建议。

在诸如保险公司等这样有着成千上万计算机用户的公司里，他们有自己的内部客户满意度基准。这些研究可能也与数据中心基准有关联。

软件使用情况基准 随着计算机软件成为重要的商业和运营工具，很显然，软件的使用往往能够改善各种知识工作和行政工作的绩效。事实上，在计算机技术出现之前，保险公司需要雇用数以百计的文书工作人员来处理保险申请、理赔以及其他文书工作。当前，大多数此类工作已被计算机软件所取代，由此导致保险公司的雇员结构发生了显著变化。

就像功能点度量指标可以用来度量软件产品一样，功能点指标还可以用来度量软件的消费情况。虽然软件使用情况基准在 2009 年还比较稀缺，但随着经济衰退的持续，该基准的重要性很可能会快速增长。

例如，针对软件项目经理所做的软件使用情况基准研究表明，与完全手工进行项目估算和规划的经理们相比，那些配备了约 3000 个功能点的成本估算工具和 3000 个功能点的

项目管理工具的项目经理们，他们的软件项目失败的更少，而项目进度更快。

软件使用情况研究还表明，配备有良好软件工具的很多知识工作者要明显胜过那些没有装备任何软件的同事。在法律、医药以及工程等知识工作方面确实如此，而在市场营销、客户支持和软件维护等数据扮演着重要角色的工作上同样如此。

软件消费基准研究从2009年才刚刚兴起，但10年之内，尤其是如果经济衰退不断持续，该基准研究很有可能会成为主流软件基准之一。

软件诉讼与失败基准 在违反合同、软件质量不佳、欺诈、成本超支或项目失败等的诉讼案件中，软件基准扮演了重要的角色。在这类案件中，常常会雇用软件专家证人，为诸如质量控制、进度、成本等相关主题的行业标准预备相关报告或者出庭作证。对于税务案件，如果诉讼案件牵涉软件资产的价值或置换成本，行业专家也会被邀请参与案件工作。

在为诉讼案件准备的专家报告中，通常会将案件涉及的具体情况与诸如缺陷去除效率水平、进度、生产力、成本等主题的行业背景数据进行比较。

每个软件诉讼搜集到的数据都是独一无二的，其中一个关键方面是软件失败的起因。大多数公司的内部软件项目失败都不会闹上法庭。但根据合同开发软件的失败闹上法庭的频率则高得多。这些案件都会有广泛而深入的举证和证人出庭作证阶段，所以在这些案件中工作的专家证人能够接触到其他任何来源所无法获得的独特数据。

根据诉讼案件获得的软件基准可能是为什么软件项目会被终止、进度延期、超出预算或者在发布之后仍有过度数量缺陷等数据的最完整来源。

奖励基准 很多组织会为那些在某些方面表现卓越的公司或个人给予奖励。比如，众所周知的波多里奇国家质量奖就是为质量与客户服务而设立的。“福布斯”年度奖颁给全球前100家最优秀的公司，这是对这些公司的另一种奖励。J.D. Power and Associates公司则会为在各种客户服务和客户支持工作中表现卓越的组织颁发奖项。对于那些追求同行业最佳的公司，需要开展涉及波多里奇奖标准的特殊类型基准研究。

如果某个公司是某些奖项的候选者，那么在收集必需的基准信息上，需要花费相当多的功夫。不过，只有那些实际上业绩很好、相当成熟的公司才有可能具有这笔开支。

截至2009年，各种企业、政府团体及软件杂志所颁发的奖项大概至少有十几种。这些奖项是为客户服务、高质量、创新性应用以及其他很多主题而设立的。

6.4.2 软件基准研究的类型

对于软件基准研究有很多方法可用来收集数据。这其中包括通过邮寄邮件或发送电子邮件而进行的调查问卷、现场访谈以及两者相结合的方式。

根据基准研究期间参与者是否知道其他提供数据和信息的参与者，基准研究又可分为“公开”或“不公开”等类型。

公开基准 在一个完全公开的基准研究中，所有参与组织的名称都是公开的，而这些参与者所提供的数据也完全公开。这种类型的基准研究很难在两个竞争对手之间开展，它通常只用于大型企业里不同部门或者不同地点的内部基准研究。

出于公司政治考虑，企业内的单独业务部门可能会对公开基准研究持抵制态度。当年

首次开始实施软件基准研究时，IBM 共有 26 个软件开发实验室，而每个软件开发实验室都声称“我们的工作太过复杂，这种基准研究可能使我们处于不利位置”。但是，IBM 仍然决定进行公开基准研究，这可谓是个英明的决定，因为它能够鼓励业务部门对他们自己的工作进行改进。

部分公开基准 公开基准研究的最常见变形之一是有限公开基准，往往只在两家公司之间进行这种基准研究。在这种只有两家公司参与的基准研究中，这两家公司均要签署极其详尽的保密协议，然后向对方提供自己公司有关方法、工具、质量水平、生产力水平、进度等方面极其详细的信息。这种研究很少能够在直接竞争对手之间开展，通常用于那些开发相似类型软件但处于不同行业公司，比如，电信公司与计算机制造公司可以彼此分享数据。

在部分公开的基准研究中，参与该基准研究的所有机构名称对所有参与者都公开，但是每个参与组织提交的具体数据则是保密的。部分公开基准研究常常在诸如保险、银行、电信等特定行业内开展。实际上，除了软件主题外，这种研究还具有多种用途，包括制定公司的薪酬及福利规划、办公室空间的安排以及人际关系和员工士气的各个方面等。

部分公开基准的一个例子是，半打位于康涅狄格州哈特福德市的保险公司的生产力和质量水平研究。所有这些公司相互竞争，但又都对“自己公司与其他公司进行比较结果会如何”很感兴趣。因此，此项研究会收集所有公司的数据，并报告每个公司与所有公司数据的平均值进行比较的结果如何。但不会提供像“哈特福德保险公司与安泰或旅行者保险公司的具体比较结果如何”这样的信息。（因为其他参与者的具体数据是保密的。）

不公开基准 在不公开基准研究中，每一个研究参与者都不知道任何其他参与者的名称。极端情况下，研究参与者可能都不知道参与该项研究的其他公司所处的行业。如果某个行业的公司非常少，或者研究的性质要求极其严格的保密措施，或者研究参与者都是相当直接的竞争对手，就需要这种水平的保密措施。

当大型企业首次开始收集基准数据时，很显然，各个业务部门的高层管理人员会比较紧张。他们都有自己的政治对手，而没有哪个高管希望自己的业务部门看起来比对手业务部门的情况糟糕。因此，每个高管都希望采用不公开的基准研究方式，这样可以隐藏具体业务部门的研究结果。但是，这是一个极其严重的错误，因为这样的话就没有人会认真对待这些数据了。

对于公司的内部基准和评估研究，最好能够公示每个部门的名称，好让企业政治充当督促业务部门改进的激励因素。这就带来了很重要的一点，即除了具有技术方面的作用，基准研究还具有政治方面的影响。

由于企业高管和项目经理经常都有自己的竞争对手，而企业政治往往又比较严重，没有人希望自己受到评判，除非他们相当确信评判结果会表明他们比平均水平优秀，或者至少比他们的主要政治对手更好一些。

6.4.3 当前的基准研究组织

目前，有相当多的咨询公司在收集各种各样的基准数据。但是，这些咨询团体往往彼

此互为竞争对手，因而很难使他们在基准信息数据上有任何种类的协调或者整合。

碰巧的是，表现较为突出的三家基准组织使用颇为类似的方式来收集软件项目活动级数据，它们是：David Consulting 集团、质量与生产力管理集团（QPMG）和软件生产力研究所（SPR）。这是因为所有这三家组织的主要负责人过去都曾在一起工作过。不过，尽管收集数据的方法很类似，他们彼此之间还是有很大差异。但是，这三者的数据总量可能是软件行业最大的基准数据集。表 6-13 列举了一些软件基准组织。

上述所有 20 个基准研究组织中，他们所使用的最主要软件度量指标是 IFPUG 功能点，其次是远远落后的第二名——COSMIC 功能点。

6.4.4 软件基准与评估数据的报告方法

一旦收集了过程评估和基准数据，紧接着的两个问题就是：谁能看到这些数据以及这些数据对什么有益？

通常，过程评估和基准研究是由那些希望改进软件性能的公司高管授权开展的。例如，有时公司 CEO 可以下令开展基准与评估研究，但更多时候是公司的 CIO 或 CTO。

软件基准与过程评估研究的一个直接用途是，向授权开展该项研究的高管展示该组织的数据与行业数据的对比结果如何。高级管理层比较感兴趣的软件主题包括如下内容：

基准内容（标准基准）

基准抽样中的项目数目

国家和行业识别码

应用软件规模

项目中所使用的方法和工具

需求变更增长率

活动级生产率

整个项目的净生产力

活动级进度情况

整个项目的净进度情况

表 6-13 软件基准研究组织举例

1.	Business Applications Performance 公司 (BAPco)
2.	Construx 公司
3.	David Consulting 集团
4.	Forrester Research 公司
5.	Galarath Associates 公司
6.	Gartner 集团
7.	信息技术指标与生产力研究所 (ITMPI)
8.	国际软件基准组织 (ISBSG)
9.	ITABHI 公司
10.	Open Standards Benchmarking Collaborative (OSBC)
11.	Process Fusion 公司
12.	质量与生产力管理集团 (QPMG)
13.	Quality Assurance Institute (QAI) 公司
14.	Quality Plus 公司
15.	量化软件管理公司 (QSM)
16.	软件工程研究所 (SEI)
17.	软件生产力研究所 (SPR)
18.	Standard Performance Evaluation 公司 (SPEC)
19.	Standish 集团
20.	Total Metrics 公司

活动级人员编制情况
 所使用的专家
 整个项目的平均人员编制情况
 活动级的工作量
 整个项目总工作量
 活动级成本情况
 整个项目的总成本
 与行业数据的对比
 基于现有数据的改进建议

组织一旦开始收集过程评估和软件基准数据，他们通常希望能够做出改进。这意味着数据收集工作是一个年度事件，而收集的数据将被作为基线以展示多年过程改进工作的进展情况。

当进行过程改进时，公司常常希望能够生成一个年度基线报告，以说明过去一年的进展情况以及下一年的改进计划。这些报告将与提交给股东的企业年度报告一起制作，也就是说，将会在下一个财年的第一季度创建这些报告。

上述年度基线报告将包括以下内容：

给企业主管和高级管理层的年度软件报告

企业业务部门的 CMMI 等级
 以类型划分的完成软件项目
 IT 应用软件
 系统软件
 嵌入式软件
 商业软件包
 其他软件（如果有）
 取消的软件项目（如果有）
 当年软件项目的总成本
 当年预算外支出
 软件诉讼
 拒绝服务攻击
 恶意软件攻击及恢复
 以软件类型划分的成本情况
 软件开发成本与软件维护成本对比
 客户满意度水平
 员工士气水平
 平均生产率
 生产率变动范围
 平均质量水平

开发期间发现的缺陷数量

产品发布后 90 天内客户报告的交付缺陷数量

当年的质量成本

本地结果与 ISBSG 及其他外部基准的对比情况

上述年度报告中的大多数数据均来自于过程评估和基准研究的结果。但是，像涉及诸如拒绝服务攻击等安全问题的一些主题，既不属于标准基准研究范畴，也不是标准过程评估的一部分，它们需要专门的研究。

6.5 总结

从大约 1969 年到 2009 年的今天，软件应用的规模和复杂性发生了极大的增长。在 1969 年，最大规模的应用软件，其规模也不超过 1000 个功能点，而在 2009 年的今天，应用软件规模动辄高达 100 000 个功能点。

在 1969 年，编程或编码工作是软件应用开发的最主要活动，占到软件项目总工作量的 90% 以上。大多数应用软件只使用一种编程语言。全世界所有的编程语言总数也不超过 25 种。在 1969 年，技术文档作家和可能的质量保证工作者几乎是当时仅有的软件专家。

2009 年的今天，编码或编程工作在大型软件项目总工作量中的比例不到 40%，而当前软件行业的专家种类也超过了 90 种。当前存在超过 700 种编程语言，而几乎每一款现代应用软件都使用了至少 2 门编程语言，某些应用软件使用的编程语言更是多达数十种。

随着软件行业在从业人员数量、应用软件规模、软件开发的复杂性等方面的增长，项目管理已严重滞后于现实需要。2009 年的今天，软件项目经理仍然在接受那些可能在 1969 年有效的培训，但这些培训达不到当今这个更复杂世界的要求。

更糟糕的是，随着经济衰退变得越来越严重，软件行业迫切需要降低软件成本。项目经理和软件工程师需要具有足够可靠的经验数据，以评估和理解与软件相关的每一项单个成本因素。不幸的是，在软件质量、安全以及成本等方面糟糕的度量实践和可靠数据的缺乏已使软件行业陷入非常糟糕的经济状况。

当今的软件行业，软件成本比几乎任何其他人造产品的成本都要高，软件产品非常容易受到安全攻击，总是充满了错误或缺陷。然而，由于缺乏可靠的基准和质量数据，无论是软件工程师还是软件项目经理都很难有效地解决这些严重问题。

软件行业需要更优的软件质量、更强的安全性、更低的开发成本和更短的开发周期。但是，直到收集了所有重要项目的可靠经验数据之后，软件工程师和项目经理才有可能为这些全行业普遍存在的问题规划出有效的解决方案。很多过程改进计划无非是采用了那些一时流行的方法，比如 2009 年时的敏捷方法，但是没有任何关于这些方法是否真正有效的可靠经验数据。更完善的度量措施和更优秀的基准研究都是软件成功的关键。

参考文献

Abran, Alain and Reiner R. Dumke. *Innovations in Software Measurement*. Aachen, Germany: Shaker-Verlag,

2005.

- Abran, Alain, Manfred Bundschuh, Reiner Dumke, Christof Ebert, and Horst Zuse. *Software Measurement News*, Vol. 13, No. 2, Oct. 2008.
- Boehm, Dr. Barry. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981. (中文版《软件工程经济学》，李师贤等译，机械工业出版社 2004 年 7 月出版)
- Booch, Grady. *Object Solutions: Managing the Object-Oriented Project*. Reading, MA: Addison Wesley, 1995. (中文版《面向对象项目的解决方案》，邢春丽、冯学民、张丽梅译，机械工业出版社 2003 年 8 月出版)
- Brooks, Fred. *The Mythical Man-Month*. Reading, MA: Addison Wesley, 1974, rev. 1995. (中文版《人月神话》，UMLChian 翻译组汪颖译，清华大学出版社出版)
- Bundschuh, Manfred and Carol Dekkers. *The IT Measurement Compendium*. Berlin: Springer-Verlag, 2008.
- Capability Maturity Model Integration. Version 1.1. Software Engineering Institute, Carnegie-Mellon Univ., Pittsburgh, PA. March 2003. www.sei.cmu.edu/cmmi/
- Charette, Bob. *Application Strategies for Risk Management*. New York: McGraw-Hill, 1990.
- Charette, Bob. *Software Engineering Risk Analysis and Management*. New York: McGraw-Hill, 1989.
- Cohn, Mike. *Agile Estimating and Planning*. Englewood Cliffs, NJ: Prentice Hall PTR, 2005. (中文版《敏捷估计与规划》，宋锐译，清华大学出版社 2007 年 7 月出版)
- DeMarco, Tom. *Controlling Software Projects*. New York: Yourdon Press, 1982.
- Ebert, Christof and Reiner Dumke. *Software Measurement: Establish, Extract, Evaluate, Execute*. Berlin: Springer-Verlag, 2007.
- Ewusi-Mensah, Kwaku. *Software Development Failures*. Cambridge, MA: MIT Press, 2003.
- Galarath, Dan. *Software Sizing, Estimating, and Risk Management: When Performance is Measured Performance Improves*. Philadelphia: Auerbach Publishing, 2006.
- Garmus, David and David Herron. *Function Point Analysis—Measurement Practices for Successful Software Projects*. Boston: Addison Wesley Longman, 2001. (中文版《功能点分析——成功软件项目的测量实践》，钱岭、苏薇、盛铁阳译，清华大学出版社 2003 年 12 月出版)
- Garmus, David and David Herron. *Measuring the Software Process: A Practical Guide to Functional Measurement*. Englewood Cliffs, NJ: Prentice Hall, 1995.
- Glass, R.L. *Software Runaways: Lessons Learned from Massive Software Project Failures*. Englewood Cliffs, NJ: Prentice Hall, 1998.
- Harris, Michael, David Herron, and Stacia Iwanicki. *The Business Value of IT: Managing Risks, Optimizing Performance, and Measuring Results*. Boca Raton, FL: CRC Press (Auerbach), 2008.
- Humphrey, Watts. *Managing the Software Process*. Reading, MA: Addison Wesley, 1989. (中文版《软件过程管理》，高书敬、顾铁成、胡寅译，清华大学出版社 2003 年 3 月出版)
- International Function Point Users Group (IFPUG). *IT Measurement — Practical Advice from the Experts*. Boston: Addison Wesley Longman, 2002. (中文版《IT 度量——专家实践》，方德英译，清华大学出版社 2003 年 12 月出版)
- Johnson, James, et al. *The Chaos Report*. West Yarmouth, MA: The Standish Group, 2000.
- Jones, Capers. *Assessment and Control of Software Risks*. Englewood Cliffs, NJ: Prentice Hall, 1994.
- Jones, Capers. *Estimating Software Costs*. New York: McGraw-Hill, 2007. (中文版《软件项目估计》，刘从越、

郝建材、申冬凯译, 电子工业出版社 2008 年 3 月出版)

- Jones, Capers. *Patterns of Software System Failure and Success*. Boston: International Thomson Computer Press, December 1995.
- Jones, Capers. *Program Quality and Programmer Productivity*. IBM Technical Report TR 02.764, IBM. San Jose, CA. January 1977.
- Jones, Capers. *Programming Productivity*. New York: McGraw-Hill, 1986.
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Boston: Addison Wesley Longman, 2000. (中文版《软件评估、基准测试与最佳实践》, 韩柯等译, 机械工业出版社 2003 年 4 月出版)
- Jones, Capers. "Software Project Management Practices: Failure Versus Success." *CrossTalk*, Vol. 19, No. 6 (June 2006): 4-8.
- Jones, Capers. "Why Flawed Software Projects are not Cancelled in Time." *Cutter IT Journal*, Vol. 10, No. 12 (December 2003): 12-17.
- Laird, Linda M. and Carol M. Brennan. *Software Measurement and Estimation: A Practical Approach*. Hoboken, NJ: John Wiley & Sons, 2006.
- McConnell, Steve. *Software Estimating: Demystifying the Black Art*. Redmond, WA: Microsoft Press, 2006.
- Park, Robert E., et al. *Checklists and Criteria for Evaluating the Costs and Schedule Estimating Capabilities of Software Organizations*. Technical Report CMU/SEI 95-SR-005. Pittsburgh, PA: Software Engineering Institute, January 1995.
- Park, Robert E., et al. *Software Cost and Schedule Estimating—A Process Improvement Initiative*. Technical Report CMU/SEI 94-SR-03. Pittsburgh, PA: Software Engineering Institute, May 1994.
- Parthasarathy, M.A. *Practical Software Estimation—Function Point Metrics for Insourced and Outsourced Projects*. Upper Saddle River, NJ: Infosys Press, Addison Wesley, 2007.
- Putnam, Lawrence H. and Ware Myers. *Industrial Strength Software—Effective Management Using Measurement*. Los Alamitos, CA: IEEE Press, 1997.
- Putnam, Lawrence H. *Measures for Excellence – Reliable Software On Time, Within Budget*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall, 1992.
- Roetzheim, William H. and Reyna A. Beasley. *Best Practices in Software Cost and Schedule Estimation*. Saddle River, NJ: Prentice Hall PTR, 1998.
- Stein, Timothy R. *The Computer System Risk Management Book and Validation Life Cycle*. Chico, CA: Paton Press, 2006.
- Strassmann, Paul. *Governance of Information Management: The Concept of an Information Constitution*, Second Edition. (eBook). Stamford, CT: Information Economics Press, 2004.
- Strassmann, Paul. *Information Payoff*. Stamford, CT: Information Economics Press, 1985.
- Strassmann, Paul. *Information Productivity*. Stamford, CT: Information Economics Press, 1999.
- Strassmann, Paul. *The Squandered Computer*. Stamford, CT: Information Economics Press, 1997.
- Stukes, Sherry, Jason Deshoretz, Henry Apgar, and Ilona Macias. *Air Force Cost Analysis Agency Software Estimating Model Analysis*. TR-9545/008-2. Contract F04701-95-D-0003, Task 008. Management Consulting & Research, Inc. Thousand Oaks, CA. September 30 1996.
- Stutzke, Richard D. *Estimating Software-Intensive Systems*. Upper Saddle River, NJ: Addison Wesley, 2005.
- Symons, Charles R. *Software Sizing and Estimating—Mk II FPA (Function Point Analysis)*. Chichester, UK:

John Wiley & Sons, 1991.

Wellman, Frank. *Software Costing: An Objective Approach to Estimating and Controlling the Cost of Computer Software*. Englewood Cliffs, NJ: Prentice Hall, 1992.

Whitehead, Richard. *Leading a Development Team*. Boston: Addison Wesley, 2001.

Yourdon, Ed. *Death March—The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*. Upper Saddle River, NJ: Prentice Hall PTR, 1997. (中文版《死亡之旅》，周浩宇、杨华译，机械工业出版社 2012 年 1 月出版)

Yourdon, Ed. *Outsource: Competing in the Global Productivity Race*. Englewood Cliffs, NJ: Prentice Hall PTR, 2005.

需求、业务分析、架构及设计

7.1 引言

在进行代码开发前，有必要先对所开发应用的特点、范围、结构以及用户界面进行定义，同时也有必要定义出用何种方法来开发这些应用的功能，以及所应用的平台。另外，还必须要定义软件应用在性能、安全性、可靠性以及其他一些方面的目标。应将所有的这些需要定义的事项逐步扩充为一系列文档，涉及需求、业务分析、架构以及设计。这里的每一项都可以分解为若干主题部分或单独的文档。

尽管针对以上各种类型的文档都存在无数的模板，但至今仍无法证明其中任何一种方法是完全行之有效的。即使是在软件工业已经发展了 60 多年的今天，在几乎所有的重要软件项目中仍然存在以下一些普遍问题：

1. 需求总是不断增长和变更，这种情况的发生每个自然月超过 1%。
2. 几乎没有任何一个软件应用能够在第一个版本中开发出 80% 以上的用户需求。
3. 部分需求是非常危险的甚至是“有害”的，因此不应该包含在产品中。
4. 部分软件应用提供了用户不需要的无关功能。
5. 大多数软件应用在安全性上非常脆弱，存在很多漏洞。
6. 在需求 and 设计中存在的错误会导致非常严重的缺陷。
7. 很少有人使用需求审查、设计审查等避免错误的非常有效的方法。
8. 标准的、可重用的需求和设计并没有广泛使用。
9. 很少有人已在有的遗留应用中进行挖掘以找到那些曾经被忽略的业务需求。
10. 文档过大以至于人们无法准确理解。

以上 10 类问题是软件工业所特有的。不同于其他实体结构的设计，如飞机、船舶、建筑或者医疗设备，软件应用的设计并不使用那些被证明行之有效的设计方法或者标准的文档格式，换句话说，如果我们阅读两个不同产品的需求文档或产品说明书，我们很可能会发现它们在内容和格式上都完全不同。这些差异使得需求与设计的检验变得非常困难，因为这些文档缺少标准和通用的格式，因而很容易被修改而产生变化，也为识别文档中的错误设置了一定的困难。自动化的需求验证和设计验证在理论上是可行的，但截至 2009 年，技术上我们还没有达到这个水平。对于需求文档及其他文档的正式审查是很有效的方法，

但很显然手工审查比起自动化方法来说要慢得多。

同时,我们也有很多种“语言”来描述需求及功能的设计。这些“语言”包括用例、用户故事、决策表、鱼骨图、状态变换图、实体关系图、可执行语言、自然语言、统一建模语言(UML),以及其他30多种各具特点的图形描述方法,如流程图、N-S图(Nassi-Schneiderman Diagram)、数据流图、HIPO图(Hierarchy Plus Input/Processing/Output, 分级结构加输入、处理和输出图)等。对于质量方面的需求,我们还有一些专门和质量功能展开(QFD)相关联的图表描述方法。

存在如此多的描述方法表明至今仍然不存在一个完美的描述方法。如果其中任何一种方法明显强于其他方法的话,那么毫无疑问,这个方法将成为现今所有软件项目所采用的实际标准方法。但截至目前我们能确定的是,任何一种方法的采用率都没超过10%。事实上,大多数软件开发项目混合使用了多种描述方法,因为没有任何一种能够达到所有业务和技术目标。因此,混合使用文本描述方法和图表描述方法(如用例、流程图及其他图表)是最常见的方法。

在本章中,我们将要讨论处理软件需求、业务分析、架构及设计的方法以及其中的一些变化。

7.2 软件需求

如果软件工程想要成为一个真正的行业而不是一种艺术形式的话,那么软件工程师们有责任帮助用户使用一种完善、详尽并且行之有效的方法来定义用户需求。

一个专业的软件工程师应该坚持使用有效的需求收集方法,如联合应用设计(JAD)、质量功能展开(QFD)及需求审查,并将其视为自己的职责。软件工程师也有责任针对潜在的有害的需求对用户做出提醒。

一个非常常见的现象就是软件需求文献通常是消极的并做了一个不正确的假设,就是假设用户可以百分之百有效地识别需求。这是一个非常危险的假设,用户识别的需求从来都不是完整的,并且经常是错误的。如果软件项目想要成功,需求必须通过一种非常专业的方式进行收集和分析,而软件工程就是一个必须知道该如何去做好这些的专业。

软件工程师们有责任坚持使用合适的需求分析方法,这些方法包括遗留应用的数据挖掘、联合应用设计(JAD)、质量功能展开(QFD)、原型分析及需求审查。对需求分析有帮助的方法还有:让用户参加到开发团队中(对于敏捷开发而言),使用用例也是一种推荐的方法。

软件产品的用户并不是软件工程师,因此我们不能期望他们知道如何用最佳方式来表述和分析需求。所以,我们需要确保以专业水准所做的软件需求收集和分析结果移交到软件开发团队。

在2009年,几乎一半左右的软件应用都是对遗留应用进行改造,这些遗留应用有些已经使用了超过25年。不幸的是,这些遗留应用极少有可用的软件说明书或需求文档。

由于遗留应用缺乏对特性和功能进行描述的资料,一种新的需求分析方法出现了。这

种新的方法从对遗留应用的数据挖掘入手,从而提炼出其中的业务规则和逻辑。由于这种方法的出现,数据挖掘也可以用来评估软件的功能点和代码语句的规模。

7.2.1 软件需求的结构和内容

软件需求,顾名思义,它描述了一个软件应用将会包含的主要特性及功能,而同时,需求说明书还服务于其他商业用途,例如,软件需求中还需要讨论一些软件应用的局限性和约束,例如,软件应用的性能指标、可靠性指标、安全性指标以及其他类似标准。

一个软件应用将会实现的总需求数量决定了这个软件的规模,而规模会在很大程度上影响应用的开发周期和费用,因此,需求是确定软件应用规模的首要因素。

幸运的是,软件功能点估算标准的结构很好地匹配了软件需求需要考虑的基础问题。按照时间先后的顺序排列下来,软件开发团队需要对下面7个基础性的问题进行探讨,并作为收集需求过程的一部分:

1. 软件应用将会产生的输出数据
2. 软件应用所需要的输入数据
3. 软件应用必须要维护的逻辑文件
4. 所有将会成为软件应用的逻辑文件的实体对象及其相互关系
5. 软件应用应使用的审查类型
6. 软件应用和其他系统的交互界面
7. 软件应用应展现的主要逻辑法则

以上7点中,有5点是国际功能点用户组(International Functional Point User Group, IFPUG)所定义的功能点估算法中的基本元素。第4点“实体对象和关系”是British Mark II功能点估算方法以及之后的COSMIC功能点估算的一部分。第7点“逻辑法则”是功能点估算方法中的一个标准因素,为IFPUG所定义的5个基本的功能点因素添加了关于逻辑法则的考虑。

需求收集过程需要考虑的问题与功能点估测方法中要考虑的问题有很多相似的地方,这些相似性使得由需求确定功能点数量成为了一个非常明确的任务。事实上,自动化的由需求来计算功能点规模已经具有了实验形式的应用,虽然这种技术还没有得到普遍应用。

此外,在需求收集阶段,还要讨论另外30个问题,其中一些是非功能性需求,还有一些是商业上的需求以确定是否对该软件应用提供资金支持。这些额外的问题包括:

1. 软件应用在功能点和源代码方面的规模
2. 软件应用从需求到交付的进度安排
3. 软件应用开发团队的人员组成,包括核心专家成员
4. 软件应用过程中每项活动所需的费用,或者说开发一个功能点所需的费用
5. 软件应用的商业价值及投资回报
6. 软件应用的非财务价值,如市场竞争力、用户忠诚度
7. 软件应用所面临的主要风险,如意外中止、项目延期或过度开发
8. 竞争对手的产品特性

9. 应用的交付方式，如面向服务的架构（SOA）、软件即服务（SaaS）、光盘交付、网络下载等等

10. 软件应用的供应链，或者相关的上游和下游应用
11. 源于被取代的历史软件的遗留需求
12. 会影响到软件应用的法律法规（如，税法或者涉及隐私保护方面的法律）
13. 应用的质量等级，比如缺陷数量、可靠性或易用性标准
14. 应用的容错功能以防止用户出现使用错误或者意外断电等情况
15. 应用的授权许可及对于授权声明的回复
16. 软件应用运行的硬件环境要求
17. 软件应用运行的软件环境要求，如操作系统或者数据库软件
18. 软件的本地化标准，或者软件的非本国语言版本的数量
19. 软件的安全性指标及相关的数据库
20. 软件的性能指标，如果有的话
21. 针对软件的培训需求或者使用指南
22. 软件开始使用和初始化前的安装过程
23. 软件的可重用性指标，既包括其他产品可能应用的各种文档，也包括后续的产品是否会重复使用某些功能
24. 期望用户通过所开发的应用能够完成的主要任务
25. 软件应用中信息走向的控制流程或顺序
26. 未来的后续版本可能出现的需求
27. 一些潜在的有害需求的危险等级
28. 软件应用部署之后的生命周期期望，也就是产品的期望服务年限
29. 软件应用所计划的总体拥有成本（TCO）
30. 软件应用新版本或者修复版本的发布频率（年度、月度等等）

以上的 7 个主要方面以及 30 个附加的问题并不是需求阶段需要考虑的全部，但是上面任何一个方面都不应该被忽略，因为任何一方面都会对软件项目产生显著的影响。

这 37 个方面中的大部分在许多不同种类的软件产品中都是需要考虑的：商业产品、内部产品、外包产品、国防项目、系统软件以及嵌入式应用等。

7.2.2 软件需求的统计学分析

通过对上百家公司的上千个软件应用项目进行分析后，笔者发现了一些关于软件需求方面的基本事实。

随着软件应用变得越来越大，软件需求的规模也与日俱增，然而，软件需求的增长速度并不能和软件应用本身的增长速度同步，因而造成的结果就是，软件应用规模越大，软件需求越不完善。

大型软件应用需求的不完善导致了这样一个现象，那就是软件需求会持续地以每个月 1% ~ 3% 的速度发生变化。

需求中可能会包含很多的缺陷或问题，这些问题很难在软件测试阶段被解决，但是其可以通过正式的需求审查手段来发现。

需求会转化为软件的设计，而设计又会转化为软件的代码，笔者一项针对 IBM 公司的调查发现，在以上每个转化阶段，有大约 10% 到 15% 的需求没有进入到下一个阶段而被丢弃，至少在每个阶段的最初是这样的。

除了由用户引发的需求蔓延（这些需求大概都会有些商业价值），还有一些数量惊人的功能变化是由开发人员未经过任何的需求收集而添加的，这些变化有些甚至看起来显然对于用户没有什么用处。对于很多软件来说，有超过 7% 的功能是在用户不知情的情况下被开发人员添加进来的。开发人员主动提供或者无意识地提供一些功能上的变化这个问题很少在需求文档里面讨论到，当开发人员被问到为什么这样做时，最常见的回答是：我觉得这个也许会有用。

总体上说，大约有 15% 的初始用户需求没有出现在正式的需求文档中，而是在此之后作为需求蔓延而添加进来。在每个由需求到其他交付物如设计或者代码的转换点，大约有 10% 的需求被无意中丢弃而不得不在之后再次添加进来或者只能推迟到后续版本。前面提到过，开发人员会在未经用户请求的情况下擅自添加一些功能，甚至有时候是在完全不理解用户的需求的情况下。在最终交付的产品中，大约有 7% 的功能是在缺少用户需求的情况下由开发人员主动添加的，尽管其中的某些功能日后被证明确实是有用的。除了需求的计划外增加以及意外的丢弃，软件需求中还存在一些危险的或有害的需求，这些需求中大部分都包含不同严重程度的错误。

理论上讲，某些类型的需求（比如使用可执行语言描述的需求）可以使用静态分析的方法或其他自动化审查方法，但目前这种方法还只是处于实验阶段。

有些软件需求可能是有害的，如果不移除掉的话会对产品造成严重的损伤。著名的千年虫问题^①就是一个毒性需求的例子。另一个毒性需求的例子是，为了加快金融系统运行而使用的文件管理协议，如果系统的备份文件会被打开而不是存储在系统中，那么数据的完整性就无从保证。在很多系统中存在的一个常见的毒性需求，就是不能正确处理有三个字的人名^②。还有一个毒性需求存在于很多软件产品中，即不完善的错误处理程序，这个缺陷已经成为病毒感染或间谍软件入侵的首选途径。概括来讲，因为有如此多的毒性需求，因此把“完全遵照需求”作为软件质量的要求，已经不再适用了。

此时，我们可来研究一下关于软件需求的规模，以及软件需求中可能包含的问题或缺陷数量的信息。

表 7-1 从每个功能点所对应需求页数的角度展示了软件需求的近似规模。这里使用的估算方法是国际功能点用户组（IFPUG）所定义的 4.2 版本的计算方法。表 7-1 展示了五种不同的需求描述语言所对应的情况。

需要注意的是，对于表 7-1 以及本章中的其他表格，使用“用户故事”描述的需求对于

① Y2K，即 Year 2000 问题，又名千禧危机，千年问题。——译者注

② 英语国家很多人的名字由三个名字组成：first name, middle name, last name。——译者注

功能点数在 10 000 到 100 000 这个区间的软件是没有可用数据的，这是因为敏捷开发模式不会应用在如此大型的项目当中，至少没有任何数据报告给收集这类数据的组织。

表 7-1 每个功能点所对应需求文档的页数

功能点数量	自然语言	可执行语言	用例	UML 图表	用户故事	平均值
10	0.40	0.35	0.50	1.00	0.35	0.52
100	0.50	0.45	0.60	1.10	0.40	0.61
1000	0.55	0.50	0.70	1.15	0.45	0.67
10 000	0.40	0.45	0.60	0.80	0.00	0.56
100 000	0.30	0.40	0.50	0.75	0.00	0.49
平均值	0.43	0.43	0.58	0.96	0.40	0.56

表 7-1 展现的一个重要事实就是需求的规模在 1000 个功能点左右达到最大。对于更大型的项目而言，文档编写的工作量会飞速增长以至于如果所有需求都写入文档的话，那么这个文档会变得很难阅读。

表 7-2 对于表 7-1 的内容做了一定的扩展，它向我们展现了使用以上 5 种语言描述的软件产品大致的总需求文档页数。

表 7-2 不同规模的软件所产生总需求文档页数

功能点数量	自然语言	可执行语言	用例	UML 图表	用户故事	平均值
10	4	4	5	10	4	5
100	50	45	60	110	40	61
1000	550	500	700	1150	450	670
10 000	4000	4500	6000	8000	0	4500
100 000	30 000	40 000	50 000	75 000	0	48 750
平均值	6921	9010	11 353	16 854	165	8860

正如我们所见的，大型软件项目有着规模惊人的需求文档，而这些文档仍然是不全面的。事实上，完全阅读一个超过 100 000 个功能点的大型软件的需求文档需要耗费超过 2500 天的时间，或者说超过 7 年的时间！显而易见，如此大量的文档几乎是无法管理的。

表 7-3 继续扩展了我们从表 7-2 得出的逻辑，它向我们展示了不同规模的软件最终的需求完成度。

表 7-3 不同规模软件的需求完成度

功能点数量	自然语言	可执行语言	用例	UML 图表	用户故事	平均值
10	98.00%	99.00%	96.00%	99.00%	93.00%	97.00%
100	95.00%	96.00%	95.00%	97.00%	90.00%	94.60%
1000	90.00%	93.00%	90.00%	95.00%	87.00%	91.00%
10 000	77.00%	90.00%	82.00%	90.00%	0.00%	84.75%
100 000	62.00%	83.00%	74.00%	80.00%	0.00%	74.75%
平均值	84.40%	92.20%	87.40%	92.20%	90.00%	88.42%

从表 7-3 可以看出，随着软件规模的增加，软件产品的需求完成度在不断降低。这就

解释了为什么需求蔓延是软件产品工业所独有的特点。我们甚至怀疑是否存在一种需求分析方法或者需求描述语言能够真的在大型软件上实现需求的 100% 完成度。

表 7-4 向我们展示了使用不同的语言所描述的不同规模的产品中, 每个功能点中所包含的需求方面的缺陷数量。

表 7-4 每个功能点中所包含的需求方面的缺陷

功能点数量	自然语言	可执行语言	用例	UML 图表	用户故事	平均值
10	0.52	0.46	0.65	1.30	0.48	0.68
100	0.57	0.50	0.80	1.46	0.53	0.77
1000	0.60	0.55	0.98	1.61	0.63	0.87
10 000	0.70	0.60	1.20	1.60	0.00	1.03
100 000	0.72	0.65	1.10	1.65	0.00	1.03
平均值	0.62	0.55	0.95	1.52	0.55	0.88

随着软件产品的规模越来越大, 软件产品需求说明书的规模会逐渐变小, 但同样的情况并不会出现在需求中包含的缺陷或错误数量上。软件产品的规模越大, 需求中就越可能发现更多的缺陷和问题。

同时需要注意的是, 这些表格展现的仅仅是平均情况下的结果。许多预防软件缺陷的方法, 如联合应用设计 (JAD)、原型分析或正式审查可以帮助将这些典型的结果降低 60% 以上。

表 7-5 进一步扩展了表 7-4 的内容, 它向我们展示了不同规模的软件产品中可能包含的需求方面缺陷的大致数量。对于大型软件产品, 表格中所展示的缺陷数量是惊人的, 因而迫切需要使用现有的缺陷预防方法和缺陷去除方法。

表 7-5 不同规模软件产品的需求缺陷数量

功能点数量	自然语言	可执行语言	用例	UML 图表	用户故事	平均值
10	5	5	7	13	5	7
100	57	50	80	146	53	77
1000	600	550	980	1610	630	874
10 000	7000	6000	12 000	16 000	0	10 250
100 000	72 000	65 000	110 000	165 000	0	103 000
平均值	15 932	14 321	24 613	36 554	229	22 842

需要注意的是, 表格中的缺陷数量是包括了所有严重等级的, 这些缺陷中只有很小的一部分会引发严重的问题。但是考虑到需求中潜在的成千上万的缺陷, 很明显地, 在超过 1000 个功能点的软件产品中, 对于需求文档使用正式审查及其他缺陷去除方法应该作为一个标准的实践惯例。

由于表 7-5 中的数字是如此巨大和惊人, 表 7-6 只展示了那些可能发生的非常严重或者有巨大影响的缺陷。

表 7-6 中所包含的缺陷都是非常严重的, 它们会对用户的使用造成非常恶劣的影响, 而问题一旦发生, 又要投入巨大的花费去修复, 这方面的一个例子就是千年虫问题。

表 7-6 引发严重影响的毒性需求数量

功能点数量	自然语言	可执行语言	用例	UML 图表	用户故事	平均值
10	0	0	0	0	0	0
100	0	0	0	0	0	0
1 000	1	1	2	4	1	2
10 000	15	14	25	40	0	19
100 000	175	150	300	400	0	205
平均值	38	33	65	89	0	45

概括来讲，对于超过 10 000 个功能点的大型软件来讲，需求收集是不可能做到面面俱到的，至少迄今为止从未全面过。

另外，在需求中还存在缺陷，而这些缺陷中的一小部分会引发非常严重的问题。因此我们需要在需求缺陷研究、缺陷预防和缺陷去除等方面进行深入探讨。

一个需要进行深入研究的话题就是，到底需要多少人参与到需求收集过程中来。通常情况下，客户的负责范围大约可以覆盖 5000 个功能点，这个数字反映了一个用户通常能够准确定义其需求的软件功能点平均数量，一般来说用户能够负责的功能点数的范围大约是从 1000 个功能点到 10 000 个功能点。

系统分析人员或者业务分析人员的负责范围会更大一点，最多可以胜任 50 000 个功能点，但平均的数量大约是 15 000 个。

以上三种不同的负责范围意味着一个大约有 50 000 个功能点左右的大型软件项目需要一个系统分析人员与大约 10 名客户进行访谈。换句话说，分析人员和客户的比例大约是 1 : 10。

这个比率对于使用开发团队内部用户来定义需求的敏捷开发模式有另外一层含义，由于大部分敏捷开发的项目都比较大，而且不超过 1500 个功能点，一个用户已经足够去定义所有的需求了。然而对于大型项目来说，引入更多用户的参与是必需的。

另外一个需要探讨的话题是收集的需求和进行分析的需求的比例，如果你假设一个典型的联合应用设计（JAD）活动需要 4 个用户参与者及 2 个商业分析人员，那么他们每天大约可以讨论 1000 个功能点并形成文档。需要注意的是，如果使用自然语言，那么每个功能点大约会形成 0.5 页的需求说明书，而使用统一建模语言（UML）则会形成 0.75 页的需求说明书。

一个敏捷开发团队内部的用户每天大约可以解释 200 个功能点，用户故事通常是比较简洁的，因此每个功能点大约会形成 0.3 页需求说明书。然而，这个需求说明书是不全面的，因此对于敏捷开发的需求来说，用户和开发者之间的口头交流是必须要进行的。

7.2.3 建立可复用软件需求的分类系统

为了建立软件应用的生产率及质量的参照基准、功能特性分析及统计学分析，我们有必要记录一些软件应用的基本信息。令人吃惊的是，软件行业并没有一个标准的分类系统以使一个软件应用可以被唯一确定。为了填补这个空白，笔者研究了一套分类系统以使我

们可以对软件应用进行明确的统计学分析。

为了在统计学上确定软件的分类并研究不同行业的软件需求，我们必须要知道关于软件的一些基本信息，比如软件开发者所在的国家或地区，以及软件所应用的行业。我们使用下面的一些标准代码来记录这些信息：

国家代码 = 1 (美国)

区域代码 = 06 (加利福尼亚)

城市代码 = 408 (圣何塞)

行业代码 = 1569 (电信行业)

CMMI 等级 = 3 (可控的并可重复的)

项目开始日期 = 2009 年 4 月 20 日

计划结束日期 = 2011 年 5 月 10 日

实际完成日期 = 2011 年 9 月 25 日

以上所使用的代码来自于国际地区代码表、国际标准化组织 (ISO) 代码以及北美工业分类系统 (NAIC) 商业部门的代码表。使用这些代码并不会影响对于软件规模的计算，但却可以为建立参照基准和研究国际经济提供非常有价值的信息。这是因为软件项目的费用随着国家、地理区域以及所应用的行业的不同而千差万别。为了使历史数据能够有价值，我们有必要记录所有会影响项目费用、项目周期以及其他项目要素的因素。

“CMMI 等级”这一项是关于著名的由软件工程研究所 (SEI) 开发出来的软件能力成熟度模型的。

在制定了软件产品所属的地域及行业以后，我们再来看这个分类系统的组成。我们所说的系统由以下 7 部分组成：

1. 项目属性
2. 项目范围
3. 项目分类
4. 项目类型
5. 算法复杂度
6. 代码复杂度
7. 数据复杂度

为了在不同的软件项目之间建立对比，我们需要非常准确地知道所比较的软件属于什么类型，这通常并不像听起来那么容易。软件工业长期以来一直缺少一个标准的软件项目分类系统来清晰准确地确定软件项目所属分类。

通过使用一些多项选择的问题，这个分类系统将超过三千五百万个影响因素浓缩到了一定数量的数据项中，这些数据项可以很容易地用来进行统计分析。这个分类系统的主要目的是提供一个基础架构以提高对于软件项目进行研究和分析的能力。

从 1984 年以来，人们一直在使用此处展示的分类系统，在笔者之前的几部著作中曾经对这个分类系统进行过一些说明，这些著作包括：《Estimating Software Costs》(McGraw-Hill, 2007)、《Applied Software Measurement》(McGraw-Hill, 2008) 以及本书的早期版本和

一些专门论述软件项目分类的著作。在笔者所设计的软件项目费用估算工具中，也包含了这个分类系统。这个分类系统的组成要素包括以下几部分：

项目属性：

1. 全新开发的软件应用
2. 已有软件应用的改善和增强（对已有的软件增加一些功能）
3. 已有软件应用的维护（对已有软件的缺陷进行修复）
4. 已有软件应用的转换和移植（将已有的软件迁移到新的平台上）
5. 已有应用的再设计（遗留应用的再次应用）
6. 应用修订包（修复所购买的软件）

项目范围：

1. 运算逻辑
2. 子程序
3. 模块
4. 可重用的模块
5. 一次性的项目原型
6. 进化的项目原型
7. 子项目
8. 独立应用
9. 系统组件
10. 系统软件版本（非初始版本）
11. 全新的部门级系统（初始版本）
12. 全新的公司级系统（初始版本）
13. 全新的企业级系统（初始版本）
14. 全新的国家级系统（初始版本）
15. 全新的全球级系统（初始版本）

项目分类：

1. 个人项目，仅限于私人使用
2. 个人项目，供其他个体使用
3. 学术项目，用于学术环境
4. 内部项目，供在同一个场所的用户使用
5. 内部项目，供在多个场所的用户使用
6. 内部项目，供内部网络用户使用
7. 内部项目，由外部的承包商开发
8. 内部项目，包含分时使用的功能
9. 内部项目，包含军事上的特性
10. 外部项目，产品会用于公共领域中

11. 外部项目, 产品会用于互联网中
12. 外部项目, 产品会租借给用户
13. 外部项目, 和特定硬件绑定供应
14. 外部项目, 无绑定产品, 产品会推向商业市场
15. 外部项目, 和商业公司签署合同
16. 外部项目, 和政府签署合同
17. 外部项目, 和军方签署合同

项目类型:

1. 非程序项目 (自动生成的、查询、电子表格)
2. 批处理程序
3. 网络应用程序
4. 和用户有互动的程序
5. 为用户提供了图形化界面进行互动的程序
6. 数据库应用的批处理程序
7. 和用户有互动的数据库应用程序
8. 客户机/服务器 (C/S) 架构的程序
9. 计算机游戏
10. 科学运算或数学运算程序
11. 专家系统
12. 支持性的系统或程序, 包括中间件
13. 面向服务架构 (SOA)
14. 通信或电信项目
15. 流程控制项目
16. 可信系统 (Trusted System)
17. 嵌入式或即时通信类项目
18. 图形、动画制作或图像处理类程序
19. 多媒体程序
20. 机器人技术或机械自动化程序
21. 人工智能程序
22. 神经网络程序
23. 混合型程序 (以上多种类型的组合)

算法复杂度:

1. 算法不需要计算或仅有少量的逻辑
2. 算法主要由一些简单的计算及简单的逻辑运算组成
3. 算法主要由简单的逻辑运算组成, 但有少量中等复杂度的运算
4. 算法中简单的和中等复杂度的计算和逻辑运算并存
5. 算法主要由中等复杂度的计算和逻辑运算组成

6. 算法中既有简单和中等复杂度的逻辑运算，也包括小部分高难度的逻辑
7. 算法中高难度的逻辑多于简单的或中等复杂度的逻辑
8. 算法中绝大部分都是高难度的复杂的逻辑
9. 算法全部为高难度的逻辑，其中包含部分极其复杂的逻辑
10. 算法中的计算和逻辑全部都极其复杂

代码复杂度：

1. 大多数的“编程”可以通过点击按钮和下拉列表选择来完成
2. 简单的非过程代码（自动生成的、数据库、电子表格）
3. 同时包含简单的和中等难度的非过程代码
4. 使用项目主体框架及可复用模块来编程
5. 中等难度的软件结构及简单的模块和路径
6. 优秀的软件结构但包含一些复杂的模块和路径
7. 代码中包含部分复杂的模块、路径及不同数据段落之间的链接
8. 代码中包含较为复杂的模块、路径及不同数据段落之间的链接
9. 代码中主要的路径和模块都非常庞大且复杂
10. 非常复杂的代码结构并包含大型的模块和难以处理的链接

数据复杂度：

1. 软件应用不需要维护任何永久性的文件或数据
2. 软件应用仅需要维护一个文件并仅有很少的数据交互
3. 软件应用仅需要维护少量包含简单数据的文件
4. 软件应用需要维护多种数据元素，但数据元素间的关联很简单
5. 软件应用需要维护多个包含中等复杂度数据的文件
6. 软件应用需要维护多个文件，部分文件包含复杂的数据元素及数据交互

数据元素及数据交互

7. 软件应用需要维护多个文件，这些文件都包含复杂的数据元素及数据交互

8. 软件应用需要维护多个文件，这些文件的大部分内容都是复杂的数据元素并有大量数据交互

9. 软件应用需要维护多个文件，这些文件均为复杂的数据元素并有大量的数据交互

10. 软件应用需要维护大量包含复杂数据元素的文件并伴有复杂的数据交互

通常使用这套分类系统进行规模估算时，用户会针对系统的各个部分提供一系列的整数值，如下所示：

通常项目属性、范围、分类及类型的结果用整数表示，但我们可以允许其他三个复杂度因素的结果包含至多两位小数，因此，如下的结果是允许出现的：

项目属性	1
项目范围	8
项目分类	11
项目类型	15
算法复杂度	5
代码复杂度	6
数据复杂度	2

项目属性	1
项目范围	8
项目分类	11
项目类型	15
算法复杂度	5.25
代码复杂度	6.50
数据复杂度	2.45

借助这套系统所得到的由数字所组成的结果为我们提供了一个独特的模式，这个模式可以帮助我们对软件应用的功能和需求进行估算、测量、基准制定及统计分析。这个系统可以帮助我们很容易地预测新项目的结果，我们只要去检查在这套系统中和这个新项目有相同或类似模式的现存项目的结果就可以了。因为非常巧合的是，有着相同模式的两个软件应用在功能点规模（但不是源代码规模）方面通常有着相同的结果。

有着相同模型的软件应用不仅有着相似的规模，而且通常倾向于包含同样的功能和相似的需求。因此，我们需要像上面的例子中那样在这套分类系统中记录一个软件应用的结果，这一步可以帮助我们建立一系列可重用的需求，从而为将来的数十个甚至数百个软件应用服务。这套系统也可以帮助我们收集面向服务架构（SOA）的系统所需要的功能。

当我们把这个系统所有的元素都统计完成后，我们就会得到如下所示的结果：

国家代码	1（美国）
区域代码	06（加利福尼亚）
城市代码	408（圣何塞）
行业代码	1569（电信行业）
CMMI 等级	3（可控的并可重复的）
项目开始日期	2009 年 4 月 20 日
计划结束日期	2011 年 5 月 10 日
实际完成日期	2011 年 9 月 25 日
项目周期延迟	4.25（月）
初始估算规模	1000（功能点）
重用功能的规模	200（功能点）
未计划变更的规模	300（功能点）
最终交付产品规模	1500（功能点）
初始估算规模（源代码）	52 000（逻辑代码行）
重用代码的规模	10 400（逻辑代码行）
未计划变更的规模	15 600（逻辑代码行）
最终交付产品规模	62 400（逻辑代码行）
编程语言	65（Java）
重用代码的语言	65（Java）
项目属性	1（全新开发的应用）
项目范围	8（独立应用）
项目分类	11（专家系统）
项目类型	15（外部项目：无绑定产品）
算法复杂度	5.25（多种复杂度混合，并有部分高复杂度逻辑）
代码复杂度	6.50（多种复杂度混合，并有部分高复杂度路径和模块）
数据复杂度	2.45（低复杂度）

这个分类系统为软件应用提供了一个非常明确的模式，这个模式可以用来对历史项目数据进行分类，也可以帮助我们估算新的软件项目的规模。这是因为如果我们使用国际功

能点用户组（IFPUG）提供的功能点度量方法去估算软件应用的规模，有着相同模式的软件应用通常也会得到相同的规模估算结果。

如果有着相同模式的软件应用在生产率或质量方面存在差异，那么证明这两个项目所采取的开发方式在有效性方面存在差异，或者在开发团队能力方面存在差距。无论在任何情况下，这个分类系统都会使得我们的统计结果更加可靠，因为它避免了将风马牛不相及的事情进行比较的错误。

如果用代码行（Lines of code, LOC）进行度量，那么不同的软件应用几乎不会有同样的规模，因为我们有超过 700 种不同的编程语言可以使用，而且，相当多的软件使用了不止一种编程语言来进行程序编写。

有着相同规模的软件应用在开发周期和开发费用方面仍然可能存在很大差异，引起这些差异的原因包括开发团队的能力差异、所使用的编程语言、所使用的开发工具和开发方法、软件应用所在的行业及开发团队所处的地理位置等。虽然获取产品规模信息是对软件应用进行估算的第一步，但规模并不是唯一需要获取的信息。

我们上面提到的分类系统可以在软件应用开始收集需求之前很早就开始使用，由于该系统包含了一些当我们了解一个新软件产品时首先需要关注的信息，所以我们可能在需求收集阶段结束前几个月的时候就开始使用这个系统，甚至是在开始收集需求前。

我们也可以将这个系统应用在存在了很多年的遗留应用上，这类应用的总功能点数量是一个非常有用的信息，但通常情况下由于这类应用的需求文档和产品说明书几乎从不更新甚至已经无从得到，计算总功能点数量变成了一个不可能完成的任务。

这个系统也可以用于商业软件，实际上，可以用于任何形式的软件应用，包括传统的军方软件应用，因为我们有足够的公开或非公开的信息来完成这个系统对于这类军方应用的结果统计。

理论上讲，我们可以对该系统进行扩展，以包含其他我们感兴趣的内容，如开发方式、编程语言、开发工具、缺陷去除方法等。然而，以下两个因素使得这种扩展并不像想象的那么容易：

1. 每个月都会出现新的编程语言、开发工具和开发方式，因此会造成该系统的不稳定。
2. 相当多的应用使用了多种混合的编程语言、开发方式和开发工具。

不过，为了展示一下扩展后的该系统的样例，下面我们展示一下在基本的信息基础上添加了开发方式的一个例子：

国家代码	1（美国）
区域代码	06（加利福尼亚）
城市代码	408（圣何塞）
行业代码	1569（电信行业）
CMMI 等级	3（可控的并可重复的）
项目开始日期	2009 年 4 月 20 日
计划结束日期	2011 年 5 月 10 日
实际完成日期	2011 年 9 月 25 日

项目周期延迟	4.25 (月)
初始估算规模	1000 (功能点)
重用功能的规模	200 (功能点)
未计划变更的规模	300 (功能点)
最终交付产品规模	1500 (功能点)
初始估算规模 (源代码)	52 000 (逻辑代码行)
重用代码的规模	10 400 (逻辑代码行)
未计划变更的规模	15 600 (逻辑代码行)
最终交付产品规模	62 400 (逻辑代码行)
编程语言	65 (Java)
重用代码的语言	65 (Java)
项目属性	1 (全新开发的应用)
项目范围	8 (独立应用)
项目分类	11 (专家系统)
项目类型	15 (外部项目, 无绑定产品)
算法复杂度	5.25 (多种复杂度混合, 并有部分高复杂度逻辑)
代码复杂度	6.50 (多种复杂度混合, 并有部分高复杂度路径和模块)
数据复杂度	2.45 (低复杂度)
规模估算方法	1 (IFPUG 功能点估算法)
软件估算方法	3 (KnowledgePlan 软件工具)
软件项目报告管理	2 (自动化)
风险分析	0 (未使用)
财务价值分析	1 (使用)
无形资产价值分析	0 (未使用)
需求收集方法	1 (联合应用设计, JAD)
需求描述语言	5 (混合使用: 用例、自然语言)
质量需求	1 (QFD)
软件质量保证方法	1 (正式引入软件质量保证方法)
开发方法	3 (团队软件过程, TSP)
预先使用的缺陷去除方法	
需求审查	1 (实施)
设计审查	1 (实施)
代码审查	0 (未实施)
静态分析	1 (实施)
六西格玛方法	0 (未实施)
IV & V	0 (未实施)
自动化测试	0 (未实施)
测试阶段	
单元测试	1 (实施)

(续)

新功能测试	1 (实施)
回归测试	1 (实施)
组件测试	1 (实施)
性能测试	1 (实施)
安全性测试	0 (未实施)
独立测试	0 (未实施)
系统测试	1 (实施)
验收测试	1 (实施)

自 1984 年以来,我们一直在使用该统计系统,但它只包含一些基础信息。而以上加入了开发工具、开发语言和开发方式的扩展结果只是我们所假设的一个例子,之所以展示这个样例,是因为扩展后的系统可以帮助我们进行规模估算、基准分析、统计学习以及多种回归分析,从而帮助我们了解不同的开发方式和实践的有效性。

通过多选题的方式,我们将数以百万计的项目因素转换成数字形式的结果,从而产生了统计系统来帮助我们进行统计分析。同时,通过分析评价这些结果所形成的不同模式,我们很容易提高或者降低项目的生产率及产品质量。软件工业应该沿着其他工业如生物学、语言学、物理学、化学工业所走的路,将更多精力投入到实用分类系统的开发上。

7.3 软件需求方法论及实践

有很多种方法来收集、分析软件需求并把需求转化为软件产品,在这里我们列出了一些常用方法的描述及结果示例。各个方法的介绍是按照首英文字母排序的。

1. 引入了用户代表的敏捷开发需求

敏捷开发提出了一个非常有趣的观点,就是引入一个全职工作的用户代表角色加入到开发团队中,这些用户代表的职责就是为新的软件应用提供需求。这些需求通常都非常短小,因而可以在短时间内加入到应用中来并提供给最终用户使用。在一个典型的敏捷开发团队中,这些用户代表在每个 Sprint 大约会定义总需求的 5% ~ 10%,这大约等价于每个 Sprint 会有 40 ~ 200 个功能点。

对于一些小型的软件应用来说,一个用户代表就足以表述所有的用户需求,这个时候引入全职的用户代表是一个非常有效的方法。但是对于大型应用来说(比如 Microsoft Office),这种方法就显得不太有效了,因为大型应用往往有数百万用户,而任何一个用户都无法代表所有用户来进行需求定义。这种方法也不适用于某些特定的嵌入式系统,如燃油喷射控制系统。

引入用户代表到开发团队中是一个非常伟大的创新,但我们需要首先明白这种方法的局限性才能更好地使用它。关于这方面的更多信息可以参考后面的“焦点小组”、“遗留应用的数据挖掘”及“联合应用设计”小节。

2. 需求蔓延

在正式的需求收集阶段结束之后，再次发生需求变更是很常见的事情。但令人惊讶的是，很多软件应用项目并不知道该如何有效地处理需求变更。通过比较一个软件应用在需求收集阶段结束时的总需求功能点数，和该软件应用在交付时候的总功能点数（这个数字包括了在需求收集阶段结束后出现的需求），我们就可以测量需求蔓延的情况。测量结果显示，在随后的设计阶段，每个月大约有2%的需求发生变更，而在代码开发阶段每个月大约有1%的需求变更。在代码开发阶段过半之后发生变更的需求通常会延期到未来的版本中。

通常情况下，一个包含大约1500个功能点的应用在设计阶段每个月大约会产生30个功能点的需求增长，而在代码开发阶段每个月大约会产生15个功能点的需求增长。对于这种规模的应用来讲，通常设计阶段会持续两个月而代码开发大约需要8个月，因此在设计阶段大约会产生60个功能点的需求蔓延，这个数字在代码开发阶段是120个，也就是说共计会增加180个功能点的需求。因此，一个在需求收集阶段结束时定义为1500个功能点的软件应用，在实际交付时会成为一个1680个功能点左右的产品。

很明显，大型的软件应用因为有更长的周期安排因而会有更多的需求蔓延出现。

现在，我们来看一下同样规模的软件应用使用敏捷开发模式时的情况，这时每个Sprint大约会包含150~250个功能点，最终交付产品的规模仍然是大约1680个功能点，不同的是，这个时候产品的开发是分阶段进行的，而每个阶段只会开发一部分功能。

我们可以使用一些方法来减少未计划的需求变更和检验变更所带来的影响，这是处理需求蔓延最有效的方法。联合应用设计（JAD）、可执行语言和原型模拟可以帮助我们减少需求变更，而需求审查、变更控制委员会可以检验变更的影响。引入用户代表的敏捷开发模式会将需求蔓延的比率提高到每个月10%左右，但敏捷开发团队已经为这种变更做好了准备，因而它仍然是良性的。

在除了敏捷开发以外的模式中，需求蔓延还有另外一些问题：（1）这些需求比原始需求更有可能存在潜在缺陷；（2）这些需求会引起计划延期及费用超支；（3）对于外包的软件应用来说，这些需求经常会引起纠纷甚至法律诉讼。

3. 遗留应用的数据挖掘

截至2009年，超过半数的“新”软件应用是为了取代那些过时的遗留应用，这些遗留应用中有些已经服役了25年以上，但不幸的是，软件行业在管理这些应用的需求文档和设计文档更新方面并不严格，至少截至目前相当多的遗留应用是这种情况，因此，我们很难找到那些需要迁移到新的替代应用上的需求。

有些自动化工具可以帮助我们检查这些遗留应用的源代码，从而提取出这些代码中隐藏的需求并加以收集，之后作为替代应用的需求，同时也可以用来计算遗留应用在功能点方面的规模，进而帮助我们估算替代应用的规模。在正式的代码审查中也可以手工地提取出隐藏需求，但这将会比自动化的数据挖掘慢上很多。

4. 可执行语言

很多商业规则都可以用英语（或其他的自然语言）来描述，那么我们可以尝试使用自动

化技术来产生一种“方言”对需求进行描述，从而帮助我们分析需求。这并不是一个全新的想法，COBOL 语言曾经计划去完成类似的功能。由 Adrian Walker 博士领导的互联网业务逻辑 (Internet Business Logic) 组织，已经建立了这样一种“方言”并开发了相应的自动化技术，他们也提供了下载以及示例以方便大家试用。很多网站都会提供关于可执行语言的信息，MSDN 应该是目前所知最好的网站，对应的 URL 是 <http://msdn.microsoft.com/en-us/library/cc169602.aspx>。

尽管如此，我们仍有必要进行更多的研究和数据分析，目前我们仍然无法回答可执行语言如何表达类似于千年虫问题的毒性需求的问题，因为可执行语言并没有明确的分界线来区分毒性需求。而且，我们并没有对可执行语言同其他方法做过详细对比，比如在需求费用、需求缺陷或需求生产率方面。最后一点，目前还没有对混合使用了可执行语言及其他方法的应用进行过详细检查。

理论上，假设静态分析工具可以对可执行语言进行解析，我们可以对使用可执行语言描述的需求说明书进行静态分析，这样一来，我们就可以发现可执行语言中存在的逻辑错误以及遗漏，从而增加静态分析工具，如 Coverity、KlocWorks、XTRAN 等的使用价值。

长久以来，毒性需求及需求不完善一直是很难处理的问题，如果我们可以创造出基于可执行语言的自动化错误检测方法来对需求和设计进行检查，这种检测就可以帮助我们消除这类问题。未来，静态分析和可执行语言的结合将向我们展示出提高需求分析质量的良好前景。

5. 焦点小组

焦点小组是一个用户集合，这些用户会参与到新产品功能和特性的分组讨论中。通常情况下，根据目标产品所需的人口规模，一个焦点小组大约由 5 到 25 名成员组成。焦点小组会对产品提供建议，甚至会提供产品的工作模型或原型。

实践证明，如果一个软件产品需要满足多方面的需求或者提供多种应用方式，那么焦点小组是一种非常有效的方式。焦点小组方法的应用历史甚至比软件工业的历史都要长，在电气设备、机械装置或其他设备制造业都应用过。

在软件工业的背景下，焦点小组方法对于那些将多样性作为目标之一的软件应用尤其适用，这类应用的用户数通常都会有数百个甚至上千个。

6. 功能性需求及非功能性需求

软件需求可以分为两大类：功能性需求和非功能性需求。功能性需求是指一个软件应用想要包含的专门特性，它会增加软件应用的规模，通常情况下可以用功能点估算方法来进行度量。

非功能性需求是指用户比较关心的软件应用的限制和约束，如性能指标、可靠性等。要满足非功能性需求可能需要一定的工作量，但通常情况下这些需求不会增加应用的规模。

7. 联合应用设计

联合应用设计的概念源自 IBM 多伦多实验室，最初是用来作为收集财务软件需求的方法。通常情况下，联合应用设计的实施方法是召开一次有主持人的正式会议，会议的参与

方包括软件应用的干系人或用户，以及软件应用的架构师或设计师。会议双方会进行面对面的讨论，并使用标准的需求检查表来进行逐项讨论以确保覆盖所有相关议题。JAD会议通常在第三方提供的地点举行，一般情况下，由3到10名用户及3到10名开发人员参加。根据所讨论软件应用的规模不同，会议可能会持续2到15天，甚至更长时间。

JAD会议的方法已经应用许多年，留下了超过35年的经验和数据，同时，JAD方法也被认可为收集大型软件应用需求最有效的方法之一。使用JAD方法可以将需求蔓延降低到每个自然月大约1.5个百分点。

8. 模式匹配

如本章前文在讨论分类系统时所述，很多软件应用在功能点方面规模很小，举例来说，一些咨询人员可能会和某些行业的若干家公司合作，比如金融、保险、卫生保健或制造业，他们很快就会发现某个特定行业内的所有公司都会有一些相同的软件应用。确实，行业内软件应用的相似性正是企业资源计划系统（Enterprise Resource Planning, ERP）出现的原因，很多公司都有自己的ERP产品推向市场，比如SAP、Oracle、BAAN等公司。

尽管不同的软件应用之间会有一些相同的功能需求，截至2009年，软件行业仍然缺乏一种有效的方法来识别和重用这些需求。为了识别这些需求的典型样例及其相似性，我们有必要将软件应用的所有功能都使用一种标准方式记录下来，并为所有的主要功能特性建立完整的分类系统。

使用各种静态分析工具来识别多个不同应用中通用的需求样例是可能实现的，这也会帮助我们重用这些功能特性。但很多软件的需求描述仍然使用着30多种图表描述方式并混杂着各式各样的语言，只要这种现象还存在于软件工业，自动化的模式匹配就会是一件困难的事情，甚至是不可能实现的。

9. 原型分析

从定义的角度看，一个软件的原型是指仅仅保留了主要功能和逻辑结构的雏形。通常情况下，原型大约会保留完整软件应用规模的十分之一，这样做的原因是人们希望原型可以很快开发出来，比如，对于一个有10 000个功能点的应用来讲，十分之一的规模就意味着1000个功能点，显然，要在短时间内开发出这样一个原型是很困难的。

原型分析的方法最适用于大约有1000个功能点的软件应用，如此一来，十分之一的规模也就是仅仅100个功能点，这样的规模是很适合快速开发的。

软件应用的原型有两种：一次性的和逐渐进化的。顾名思义，一次性的原型在使用过后就会被丢弃，而另一方面，一个逐渐进化的原型会不断地添加进新功能，从而逐步地发展成为最终的产品。

在以上两种原型中，一次性的原型会更加安全。逐渐进化的原型为软件开发提供了捷径，但同时又缺乏对于原型的质量控制，这可能会导致原型出现安全缺陷、质量问题和性能问题。

这两种类型的原型都可以成功地减少需求蔓延，一般来说，使用了原型分析的软件应用，其需求蔓延会低于每个自然月1.5个百分点，或者说比不使用原型分析的应用要低一半以上。

10. 质量功能展开

和其他有效的质量控制方式一样，质量功能展开（QFD）也源自日本。大约在 1972 年，三菱公司第一次将 QFD 用于巨型远航油轮的质量需求分析。由于 QFD 图表的样子和一栋尖顶房屋很相像，QFD 有时也被称作“质量屋”。

虽然 QFD 源于制造业，但现在已经在软件行业得到了应用，QFD 主要用于嵌入式软件和系统软件，如机场或医疗设备。一些计算机公司如 HP 公司（即惠普公司——译者注）和 IBM 公司也将 QFD 应用于软件产品或硬件产品。

关于 QFD 有很多的书和报告，但由于学习如何使用 QFD 并成功部署需要超过一周的时间，因此在开始启动一个 QFD 项目之前，我们还需要更多的信息。有一个非营利性质的 QFD 研究所可以成为这些信息的来源之一。同六西格玛方法^⑤一样，QFD 借鉴了武术中的一些名词而使用了“带”这样一个系统来表示受训练者的等级，同六西格玛方法及许多武术一样，黑带是受训者能够达到的最高成就（当然，真正的武术学习者想要从初学者达到黑带等级需要付出多年的训练和实践，但达到六西格玛或者 QFD 的黑带只需要大约几个月的训练和非常少的亲身实践）。

11. 需求工程

需求工程是软件工程行业中一个相当新的话题，它试图使用正式的需求提取、分析方法，建立软件应用模型并检验需求，从而为需求收集和分析建立严格的标准，也就是说，需求工程技术仍然在发展进化中，目前还没有形成一个完整的学科。

需求工程的最佳适用对象是运行在复杂物理设备上的系统软件或嵌入式软件，其原因在于，相比于其他类型的软件，系统软件和嵌入式软件对于自身的成功运行有着更严格的要求和质量标准。

虽然截至 2009 年，关于需求工程的实践经验仍然很少，但一些非正式的证据表明，相比于随意使用了其他需求方法的类似软件应用，使用了需求工程方法的应用在某种程度上有着更少的需求缺陷和更高的缺陷去除效率。但是需要注意的是，能够使用需求工程的组织自身通常已经通过了 CMMI 三级认证，甚至是更高级别的认证，或者这本身就能够解释这种改进出现的原因。

我们可以使用需求工程方法和其他正式开发方法进行协作以相互促进，如 Rationa 统一过程（Rationa Unified Process, RUP）和 UML 方法，需求工程方法也可以和团队软件过程（Team Software Process, TSP）很好地结合使用。但需求工程方法通常不会用在敏捷开发项目中，因为需求工程的严格和敏捷开发是冲突的，我们很难对短小的用户故事进行正式的需求工程分析。

12. 需求审查

对需求等软件交付物进行正式审查的方法源于 20 世纪 70 年代的 IBM 公司，尽管已经有了超过 40 年的历史，需求审查仍然是最有效的缺陷去除方法并有着最高的缺陷去除效

⑤ 即 Six Sigma，于 1986 年由摩托罗拉公司的比尔·史密斯提出，属于品质管理范畴，旨在生产过程中降低产品及流程的缺陷数，防止产品变异，提升品质。——译者注

率。正式的需求审查最高可以达到 85% 的缺陷去除效率，并很少有低于 65% 的缺陷去除效率。与之相对应的是，各种形式的测试的缺陷去除效率通常都不会高于 35%，即使有，也几乎不会超过 50%。同时，审查对于缺陷预防也有很大作用，因为审查参与者会自觉地在后续工作中避免审查中曾经发现的同类缺陷。

审查是一项需要团队合作的活动，参与者包括定义好的主持人、记录员、审查者以及被审查对象。很多书籍都会讲到审查这个话题，大量的数据也可以帮助我们理解审查方法。2009 年，一个新的非软件审查组织成立了，以取代之前在 20 世纪 80 年代建立的软件审查检验组织（Software Inspection and Review Organization, SIRO）。

13. 需求追踪

每当我们定义好一个特定的需求时，这个需求必须要包含在设计文档和源代码中。相关的测试用例也需要创建出来以确保需求在软件应用中可以正确呈现。相应的培训资料 and 用户参考手册也需要创建出来以便指导用户如何使用软件应用。需求追踪讨论的就是从一些交付物如代码或测试用例中反推出需求的方法。

理论上讲，如果我们为每个确定的需求分配一个唯一识别码或序列号，那么双向的追踪溯源都是可以实现的，因为一旦为需求分配了识别码或序列号，那么和这个需求相关的产品说明书、代码、测试用例及其他交付物都会使用相同的数字。

我们通常使用二维坐标矩阵来进行需求溯源，矩阵的一个轴列出所有的需求，另一轴列出所有包含各个需求的文档或代码段，两个轴的交叉点表示某个特定需求是否已经实现。

理论上讲，追踪应该是正向的，但在实际应用中，尽管很多自动化的工具可以帮助我们，但需求的追踪仍然是困难的和错综复杂的。

需求追踪最经常用于国防应用、系统软件及嵌入式软件，因为这些应用本身通常都会涉及法律问题或者债务问题。随着《萨班斯-奥克斯利法案》的出台，需求追踪对于信息科技企业的软件应用也变得非常重要，因为这个法案会对向政府出售功能很差的财务软件的 500 强企业征收罚款。

需求追踪极少用于互联网应用、企业级软件或者移动设备如 iPhone 上的软件程序，同样，需求追踪也基本不会用于单一企业内部应用的开发上。

需求追踪方法仍然存在大量非常严峻的问题，这也是大多数关于需求追踪的文献所讨论的。尽管有超过 100 种工具声称可以帮助用户进行需求追踪，想要进行有效的需求追踪仍然会有很多麻烦，并且仍然没有完美的方式。

如果多个软件应用会使用同一个可重用的需求，那么非常明显，追踪不仅要在单个应用内部使用，同时也要在跨应用间进行，这样一来我们就需要一个三维坐标矩阵来进行追踪了。

14. 可重用需求

在同一个行业里，很多软件应用都会执行相似的功能，例如，不同保险公司的理赔程序是相似的，数百家公司的数千个应用程序有着类似的订单处理过程和货品计价过程，又如，几乎所有软件应用都需要容错功能。

理论上讲,如果我们假设存在一些便捷可用并且有着较高可靠性的标准功能组件,那么任何一个商业软件应用至少有 60% 到 75% 的功能是可以使用这些组件来实现的。但不幸的是,我们正是缺少这样一个可重用组件的目录。这个目录需要记载可重用需求的定义、设计、代码、界面以及测试用例。显而易见,这些通用的功能特性也需要提供回溯到原始来源地的功能,以防这些可重用功能组件中出现错误或者出现某个功能组件被召回的现象。

在某些特定行业内,如国防软件行业或航空软件行业,存在一些可重用功能组件的目录,这些可以作为我们需要建立的目录的样本。截至 2009 年,我们仍然没有一份业内通用的概括性组件目录可以使用。

碰巧的是,本章前面所讨论过的分类系统可以通过向下扩展从而来描述单独的或特定的可重用功能或特性,这是因为几乎每个软件应用所提供的功能或特性都提供相似的服务或者执行相似的动作。为了建立一个可重用功能组件的分类系统,我们需要讨论下面这些话题:

1. 此功能的起源
2. 此功能所建立的日期
3. 此功能的版本号
4. 此功能的认证等级
5. 此功能的业务目标
6. 此功能的名称
7. 此功能用于进行追踪溯源的序列号
8. 实现此功能所用的编程语言
9. 和此功能相关联的可重用测试用例的链接
10. 和此功能相关联的可重用文档的链接
11. 相关功能的链接
12. 此功能的输入数据
13. 此功能的输出数据
14. 此功能对其他软件功能的输出信息
15. 此功能从其他软件处接收到的输入信息
16. 此功能相关的实体及其相互关系
17. 此功能所使用的逻辑文件
18. 对此功能所使用的审查类型
19. 如果此功能和其他软件的交互不仅限于信息的话,那么此功能和其他功能的接口定义
20. 此功能的容错方式
21. 此功能的加密方式
22. 此功能所使用的算法逻辑

很明显,可重用需求的出现会为需求追踪扩充一个额外的维度。对某个特定应用来说,我们需要从代码对需求做反向追踪,而如果多个应用都是用同一个可重用功能组件的话,我们就需要做跨应用的需求追踪,这个时候我们会需要一个立体的三维坐标矩阵。

15. 安全需求部署

由于全球经济衰退的加剧,蠕虫、病毒、间谍软件、按键记录器及 DoS 攻击等各种形式的攻击也日益加剧。大部分的软件工程师和质量保证人员并没有在安全控制技术方面受过足够培训,因此在防范这些问题上并不太有效,而大多数的软件用户在这方面也完全无能为力。本书中介绍的安全需求部署 (Security Requirements Deployment, SRD) 的概念,借鉴了质量功能展开 (QFD) 在质量需求方面的做法,将同样严格的标准应用在安全需求上。然而,为了使安全需求部署更加有效,我们还需要讨论另外一个影响因素,既然 SRD 要讨论的是安全需求,除了开发团队和用户代表之外,我们很有必要引入一位最顶尖的安全专家来参加 SRD 的计划会议。

SRD 计划会议上需要讨论的问题包括最常用的安全保护方式,包括物理隔离及避免最常见的安全漏洞等。然而,现实情况的紧迫性要求我们必须找到更加先进的方法来提高软件代码对于外部攻击的抵抗能力,例如,提高软件逻辑运算速度、严格限制权限、使用一些编程语言如 E 语言来产生抵抗攻击的代码,等等。另外,还可以使用类似于 Google 所开发的 Caja 方法来对抗攻击。Caja 这个词语在西班牙语里面是“盒子”的意思,这个方法指的是 Google 开发的一种将 JavaScript 代码或 HTML 代码封装起来的方法,通过这个方法阻止任何外部应用对封装代码进行攻击或修改。

此外,SRD 会议还需要讨论以下问题,首先是安全审查,使用最佳静态分析工具来找到软件中的安全漏洞,其次是引入一些特殊的安全测试阶段。我们可以考虑雇用一些“有道德的黑客”来模拟对于软件应用的攻击,他们会尝试进入到软件内部进行攻击或者获取访问软件内部机密信息的权限,甚至尝试夺取软件的控制权限,这样的一种测试对于提高软件安全性能也许会有重大意义。

时至今日,软件安全漏洞的严重性需要我们立即行动起来寻找解决方案,使用防火墙并安装防病毒软件及防间谍软件已经不足以提供足够的保护了。在现今世界,对于软件的攻击不再仅仅来自于一些恶意的业余黑客,也包括一些由外国政府提供资金和培训的职业黑客以及一些资金充足的犯罪集团。

16. 统一建模语言 (UML)

UML 建模语言是 Rational 统一过程 (Rational Unified Process, RUP) 的一部分,RUP 现在属于 IBM 公司所有。UML 建模语言融合了 Grady Booch、James Rumbaugh 和 Ivar Jacobsen 三人的思想,这一点在软件社区已经广为流传。UML 语言及它的前身是针对面向对象的需求和设计而产生的,但实际上,它们几乎可以应用在任何形式的软件上。

UML 建模语言是一套内容丰富繁杂的图表表示方法,不仅包括软件需求的描述,也包括对软件架构、数据库设计及其他软件产品工艺的描述。事实上,UML 2.0 包含了 13 种不同种类的图表。也正因为 UML 的结构如此丰富,任何人在应用 UML 建模语言之前都需要一个漫长的学习过程。

截至 2009 年,大量的商业工具可以帮助我们构建和管理 UML 图表。使用标准的需求审查方法和设计审查方法,我们可以很容易地对 UML 图表进行审查,但我们仍然需要一些自动化工具来进行一致性检查和有效性检查,在这里我们能想到的就是一种类似于多种静

态分析能力的集合。

对于可能会被多种软件应用所使用的可重用需求或可重用功能组件，我们需要找到一种智能模式匹配工具来帮助我们搜索 UML 图表以便找到类似的图表模式。

UML 建模语言并不能包治百病，不过对象管理组织（Object Management Group, OMG，对象管理组织是一个国际协会，开始的目的是为分布式面向对象系统建立标准，现在致力于建立对程序、系统和业务流程的建模标准，以及基于模型的标准——译者注）正在不断地为 UML 添加新的实用功能并摒弃那些令人烦恼的元素。因此，UML 在未来应该会变得更加实用。

UML 图表是标准功能点分析方法的常用输入数据，理论上，我们完全有可能开发这样一个系统工具，通过对各种 UML 图表进行分析从而自动得出总功能点数量。事实上，建造类似的工具已经处于实验阶段了。

17. 用例

用例的概念由 Ivar Jacobsen 提出，Ivar 同时也是 UML 的先驱者之一。虽然用例和 UML 有所关联，它仍可以作为一个独立的需求收集方法并被广泛应用。用例的目标是直接地描述功能需求，它使用一种有趣的视觉展现方式向人们呈现用户使用软件时的动作，包括如何提出一个请求、修改请求、控制软件行为以及最终完成请求。软件应用自身就如同一个黑盒子一样，而用例只专注于用户如何同这个黑盒子进行交互以完成商业功能。

用例为软件需求分析引入了一些非常有趣的概念，如“操作者”、“角色”等。这些概念将我们的精力集中在本质性方面并引导分析人员和客户进入一个正确的方向。

很多模版可以帮助我们设想用户使用软件时的一系列动作，这些模版通常会包括诸如“目标”、“操作者”、“前提条件”、“触发条件”等方面。

和 UML 建模语言的特点一样，很多商业工具都可以帮助我们创建和管理用例。用例同样可用于正式的需求审查和设计审查，并且可以通过功能点分析来预测软件应用的规模。总体来说，用例是最容易进行审查的需求形式之一，因为它所采用的视觉呈现形式可以让我们非常容易地检查需求中包含的假设条件。

联合应用设计（JAD）方法中也会使用用例，有时在 JAD 会议的过程中就会创建出用例。

18. 用户故事

敏捷开发的目标是尽可能快地开发出可以使用的产品代码，敏捷开发社区认为 UML 方法所需要的大量文档系列及有时候会用到的用例都是开发进度的障碍，而不是一个有效的解决方案。因此，敏捷开发社区研究出了一种新的快速灵活收集需求的方式，用户故事便应运而生了。用户故事的一个显著特征就是它和测试用例紧密关联，事实上，用户故事和与其相关的测试用例是同时开发出来的。

为了保持用户故事的简洁性，同时保持敏捷开发尽量减少文档的哲学，用户故事通常是写在一张 3 英寸 × 5 英寸的卡片上，而不是标准办公用纸。很多用户故事非常简短，只是非常简单的一句话或几句话，举个例子，“我想从 ATM 机中取钱”。然而，这同时也意味着如果要描述一个非常复杂的事务流程，我们需要使用数十张卡片，而每个卡片只能描述整个流程中的一个步骤。

用例可以作为功能点分析的输入,但用户故事却不适宜这样做,因为它们过于简洁而缺少必要的细节描述,这也是敏捷开发中不常使用功能点分析的原因之一。事实上,对用户故事进行分析可以有一个备选方案,那就是使用相关的测试用例作为功能点分析的基础,因为测试用例必须要更加详细和完整。

由于用户故事的简洁性,通常情况下对其进行正式的审查也很难发现其中的缺陷,这种情况下,由于用户故事和相关的测试用例是同时开发出来的,我们可以对测试用例进行审查,这通常都会有一些潜在的价值。

关于用户故事的另一个问题就是它的使用期限,一旦开发团队将产品的最初版本交付给客户,后续版本的开发很可能会由另一个开发团队来承担甚至可能会外包给其他公司,那么这些继任者如何能够得到初始版本的需求呢?换句话说,用户故事是否是一个实用有效的信息传递方式,可以将需求信息保存10年甚至20年之久呢?

一些敏捷开发组织使用一种叫做故事点的度量指标来进行规模估算,然而,至今并没有任何使用故事点的大型基准数据集。另外,如果一个项目的需求源自于故事点,而其他类似项目使用其他需求方法(如UML或者用例),我们便无法将这个项目和其他项目进行对比。

将故事点转化为功能点在理论上是可行的,但如果敏捷项目能够使用任何一种高效的的功能点估算方法,这将会是一个更好的解决方案。提供功能点估算可以帮助我们将敏捷项目和其他项目进行点对点并排比较,同时也可以使得敏捷项目能够将规模数据提交给基准组织如国际软件基准组(International Software Benchmarking Standards Group, ISBSG)

19. 在2009年前后软件需求的现状小结

即使是在软件开发行业已经发展了60年的今天,各种需求收集和需求分析的方法仍然令人们感到苦恼,需求蔓延仍然广泛存在,需求缺陷以及毒性需求也时常发生。尽管需求审查可以有效地解决这些问题,但在美国国内,仅有不到5%的软件项目使用了需求审查,如果我们放眼全球,这一比例将会更低。

针对某些特定需求和特定功能,研究一套可扩展的分类系统是一个很有价值的工作,因为这个分类系统可以帮助我们对多个不同软件应用中的同类需求进行对比和评估,这是由于如果两个项目在分类系统中有着相同的模式,那么它们通常都会包含类似的需求和功能。

提高静态分析方法在软件需求分析中的利用率也是一件非常有价值的事情。同时我们可以对数据挖掘进行深入研究,以便可以从软件源代码中提取出潜在需求,而这种方法可以帮助我们应用需求静态分析方法,同样,总功能点数量的自动化估算方法也有助于静态分析的应用。

需求工程的最终目标是建立标准可重用组件的目录,这份目录同时还包括这些组件的相关测试资料和使用说明书。理论上讲,软件应用中超过50%的功能特性(这一比例甚至可能超过75%)可能最终源自经过认证的可重用资料。

7.4 业务分析

“业务分析”这个词和以前常用的“系统分析”有很多相似之处，很多企业雇用业务分析人员作为软件开发部门和公司业务部门的联络员。

由于他们的角色定位在技术人员和商业部门的联络员，业务分析人员很早就会加入到项目中来（有时甚至是在收集产品需求之前），并且他们会成为需求阶段的主要参与者。

在需求阶段结束之后，业务分析人员仍然会继续加入到软件设计和初期的产品代码开发阶段，这是因为需求蔓延会不断出现，而业务分析人员必须要对这些新的需求进行分析处理。一直到项目进入到代码开发阶段的后期，需求蔓延才会停止，因为这个时候新出现的需求会被分流到软件的未来版本。

业务分析人员的职责是协助需求收集的进行，并确保技术人员和客户方（或其他干系人）可以进行清晰有效的沟通。

截至 2009 年，对于业务分析人员的背景要求以及相关技能培训一直没有一个明确的体系。很多人在成为业务分析人员之前的工作是系统分析人员、软件工程师或者质量保证人员，他们因为想要担负起更多的责任而进入到业务分析领域。

国际商业分析研究所（International Institute of Business Analysis, IIBA）是一个非营利性组织，这个研究所一直在对业务分析知识体系（Business Analysis Body of Knowledge, BABOK）库进行维护和更新，我们可以在这个库里面找到大量关于业务分析的信息。

业务分析人员的参与对于需求收集、需求提取以及需求分析都会很有帮助，因为他们既有软件行业的背景，也有商业领域的知识背景，例如，业务分析人员通常会联合应用设计（JAD）会议的主持人。

业务分析人员还可以参与到需求审查、质量功能展开（QFD）或其他相关活动中来，无论是收集需求、分析需求还是将需求的意义解释给软件部门内的人员。

业务分析人员需要和其他方面的专业人员协同工作，因为业务分析人员在一些技能方面存在不足，下面举出了一些例子来说明业务分析人员在一些方面并不擅长：

1. 软件项目规模估算
2. 软件项目范围管理
3. 软件项目风险分析
4. 软件项目进度追踪和控制
5. 软件项目的质量控制
6. 软件项目的安全分析和防护

我们之所以敢断言业务分析人员在以上几个方面存在不足，是因为我们注意到这样一个现象，那就是无论业务分析人员是否参与到项目的需求过程中来，以上 6 个方面都普遍存在问题。

业务分析人员需要对公司和企业的软件业务有比较深入的了解。事实上，业务分析人员这个角色和我们本章后面要讨论的企业架构师角色在职责上有一定的重叠。

将来，我们需要对业务分析人员、架构师、企业架构师、项目范围管理人员、项目管

理办公室的职责范围进行清晰完整的定义,这将非常有用,因为这些人员的工作内容实际上是具有相同之处的。

业务分析人员可以从多种渠道收集并整理出各种基准数据,这对于他们的雇主来讲是非常有用的信息。事实上,对于所有的30种软件基准来说,如果能够在需求阶段的早期对其进行了解都是非常有帮助的。这30种基准包括:

1. 项目组合基准
2. 行业基准(银行业、保险业、国防工业等)
3. 国家基准(美国、英国、日本、中国等)
4. 软件应用类型基准(嵌入式软件、系统软件、信息管理软件等)
5. 软件应用规模基准(1个功能点、10个功能点、100个功能点、1000个功能点等)
6. 需求蔓延基准(需求变更的月发生率)
7. 数据中心及运行基准(可用时段、平均无故障时间^①等)
8. 数据质量基准
9. 数据库容量基准
10. 员工与专家基准
11. 员工流失率基准
12. 员工薪酬基准
13. 组织结构基准(矩阵型、小型团队、敏捷团队等)
14. 开发生产率基准
15. 软件质量基准
16. 软件安全基准(预防费用、修复费用等)
17. 售后支持及维护基准
18. 遗留应用革新基准
19. 总拥有成本(TCO)基准
20. 质量成本(COQ)基准
21. 用户满意度基准
22. 软件开发方法论基准(敏捷开发、RUP、TSP等)
23. 工具使用基准(项目管理工具、静态分析工具等)
24. 可重用性基准(各种可重用组件的规模)
25. 软件使用基准(不同功能的使用情况,不同行业的使用情况)
26. 外包基准
27. 软件项目进度延误基准
28. 费用超标基准
29. 项目失败基准(从法律诉讼的角度)
30. 诉讼费用基准

① 平均无故障时间,即 Mean Time To Failure,简称为 MTTF。——译者注

业务分析人员并不是唯一需要熟知以上基准数据的工作人员，但是由于他们在软件开发初期阶段的中心位置和重要角色，使得业务分析人员的位置非常关键，因此，他们了解得越多，他们的工作就越有价值。

每个业务分析人员的工作量范围大约是 1500 到 50 000 个功能点，这意味着项目中业务分析人员和普通软件工程师的比例大约在 1:10 到 1:25 之间，而项目中业务分析人员和客户的比例大约在 1:10 到 1:30 之间。

7.5 软件架构

本质上讲，软件架构主要关心以下 7 个方面的问题：

1. 软件应用的整体架构
2. 软件应用使用的数据结构
3. 软件应用和外界交互的界面
4. 软件应用如何分解为各功能模块
5. 信息如何在各个功能模块间传递
6. 和软件架构相关的性能特征
7. 和软件架构相关的安全特征

当然还有其他相关方面需要考虑，但是以上 7 个方面是需要关注的最基本问题。

近年来，软件架构师和企业架构师的角色定义一直在不断发展变化中，随着云计算技术、面向服务架构（SOA）和虚拟化技术的广泛应用，这种发展还将持续下去。

软件架构在项目中的重要性和房屋架构的地位有一定的相似之处，那就是：产品的结构越大，好的架构设计越重要。

实体建筑的规模是使用“平方英尺”来衡量的，而软件应用的规模是用“功能点”来计算的，巧合的是，在讨论架构的价值方面，二者共用了同一个模型。表 7-7 表明了架构的价值如何随着产品规模的增长而增加。

表 7-7 架构师的价值随产品结构规模的变化

平方英尺数或功能点数	架构的重要性
1	很难引入架构师而且也不需要架构设计师
10	不需要架构设计师
100	需要少量的架构设计
1 000	架构设计很有用
10 000	架构师很重要
100 000	架构师具有决定性作用

通过表 7-7 的数据我们可以得出，一个包含大约 5 个功能点或者说 250 行 Java 代码的小型 iPhone 应用程序，是完全不需要任何正式架构设计的，仅仅依靠开发人员个人关于结构化代码方面的知识就可以完成产品的开发。

然而，对于像 Windows Vista、Oracle、SAP 等的大型系统来说，如果没有若干架构专家和一个好的架构设计，这些软件几乎是不可能开发出来的。这些大型应用包含多达 100 000 个功能点，或者说如果用 Java 来开发的话，大约会有 500 万行代码（实际开发中可能会达到超过 1500 万行的规模）。

软件架构和建筑架构有一个相同点,那就是他们主要关注产品结构方面的问题。但是,软件架构师的工作比建筑架构更加复杂,因为软件产品在完工后并不是一成不变的。随着新功能的添加,软件产品以每年8个百分点的速度在持续增长,这显然比建筑物在完工后的变化速度要高很多。而且,软件产品只有在使用中才能创造价值,而软件产品在使用过程中,由于产品中不断的功能调用及数据的大量修改使得软件结构处于快速的动态变化中。因此,软件架构师需要考虑动态结构及软件性能相关的问题,而相比之下,建筑设计师仅仅在极少数情况下(如吊桥设计或者运输系统设计)才需要考虑这些问题。

安全方面的考虑也是软件架构和建筑架构的一个明显区别。当然,对于像五角大楼或者CIA总部这类建筑来说,安全性是一个建筑设计师要首先考虑的,但是对于大多数普通住宅或者办公室的建筑来说,安全性并不是一个主要的考虑方面。

对于软件架构师来说,软件应用安全性对于各种规模的软件都已经变得越来越重要。随着经济危机的持续,各种安全威胁变得越来越复杂,因而软件安全性也就变得更加重要。2009年初爆发的Conflicker蠕虫威胁,成功地感染了超过190万台电脑,其中甚至包括一部分处于“安全防护”下的政府机构的电脑。这次危机为我们敲响了警钟,我们必须认清软件安全问题应该作为一个架构设计问题,并提高它的重要性。

随着软件工程从一门手艺发展成为一个工业领域,软件架构的重要性也必将随之持续增长,其中一个原因就是软件架构设计风格在不断发展中。

再回到我们讨论的类比物——建筑架构上来,在历史发展中,不同时期的建筑具有不同的主导架构风格,同时,建筑设计风格又有很多地区差异。因此,在美国历史上也有很多建筑风格,例如,Queen Anne风格、General Grant Gothic风格、Southern Antebellum风格、English Tudor风格以及Frank Lloyd Wright风格等数十种。

软件工程并不像建筑设计风格那样历史悠久,但是它的变化发展速度一点也不比建筑架构慢。软件标准中有一个非常有用但却经常被忽略的信息,就是用于软件产品的架构描述或者架构分类。这些信息可以帮助我们分析各种软件架构,例如说不同软件架构的质量等级以及安全漏洞等。

对于规模不超过100个功能点的小型软件应用来说,人们对于软件架构这个话题是不太有兴趣讨论的,这基本上是20世纪60年代晚期之前的实际情况。大约在1968年,Edsger Dijkstra和David Parnas第一次提出将软件架构列为重要议题,之后还有一些先行者如Mary Shaw和David Garlan一直在不断强调软件架构是大型系统软件能够成功的决定性因素之一。

软件架构变得越来越重要主要是因为以下4点:

1. 软件应用的规模在飞速增长并已经超过了10 000个功能点或者一百万行源代码。在2009年的今天,软件规模可能已经是上述数据的十倍甚至更多。
2. 软件应用所使用数据的容量在飞速增长,其增长速度甚至可能超过了软件产品自身规模的增长速度。软件应用在执行中自动产生的数据从数以千计发展到百万级别甚至已经达到了十亿的数量级,并且还在持续发展。毋庸置疑,万亿级数据的时代即将到来。
3. 数据库技术以及数据组织结构在快速发展,其速度和软件架构的发展速度一样快,

甚至可能更快。

4. 软件应用已经不再只是单独运行在一台计算机上了。

当软件工程师开始将大型软件应用拆分成可以并行工作的不同组件，或者运行在不同的独立计算机上以便同时工作时，软件架构变成了非常重要的一个方面。

有一些软件应用将所有的功能都独立运行在一台计算机上，人们通常会认为这种软件有着“庞大的”架构设计。有一些软件并不遵循这种模式，一个例子就是将部分功能运行在主计算机（通常是大型机）上而将其余的功能运行在个人计算机上，这种模式叫做“服务器-客户端”结构。

20 世纪 80 年代和 90 年代出现了更多的软件架构方法，包括事件驱动架构（Event-Driven Architecture）、三层架构（展示层、商业逻辑层和数据库层）、比三层结构有更多层次的多层结构、点对点架构、模型驱动的体系结构以及更多最近出现的基于模式的架构，如面向服务的架构及之后出现的云计算。

在软件架构不断发展与扩充的同时，数据结构以及数据容量也在不断发展壮大。继层次性数据结构之后，出现了关系型数据结构、面向行的数据结构、面向列的数据结构、面向对象数据结构，等等。

显然，为了达到软件应用所设计的目标，软件架构师需要考虑软件自身架构以选择最优化的数据结构。这并不是无关紧要的选择，而是需要经验和专门的知识才能做出正确决定。

通过比较所有最新的软件架构和数据结构，选择出最佳组合并成功地设计出软件应用，这项工作已经成为了一个专门的职业，需要通过专门的训练和相当多的经验才能够胜任。因此，许多大型公司（如 IBM 公司和微软公司）新设立了一些工作头衔，如“架构师”和“资深架构师”。

随着架构师这个职位出现在大型公司，一些专门的协会也成立了，架构师们可以通过这些协会来分享信息，同时获得最新的一些想法。这些协会包括国际软件架构师协会（International Association of Software Architects, IASA）、全球软件架构师协会（World Wide Institute of Software Architects, WWISA）等。还有一些专门的期刊如微软软件架构师期刊（Microsoft Architecture Journal）会讨论一些软件架构问题。

截至 2009 年，无数的证据都在支持一种说法，那就是制造大型软件应用的公司应该雇用职业软件架构师，这一点应该被认为是一个最佳实践。

随之而来会有一个非常值得讨论的问题，那就是一家公司到底需要多少个软件架构师。通常一个软件架构师的工作量大约是五千到十万个功能点，这意味着对于一个规模在一万个功能点左右的应用来说，我们至少需要一个架构师；而一个规模在十五万个功能点左右的大型软件可能需要至少两个架构师。然而即使是在 IBM 或者微软这样有着超过五万个软件工程师的公司里面，软件架构师的数量可能不超过 100 个。

软件架构风格的发展速度实在是太快了，而衡量软件架构的标准又是如此的模糊，因此我们很难说对于某个特定的软件应用来说，某种特定的软件架构形式究竟是一个好的选择，还是一个可能出问题的选择，又或者是一次灾难性的选择。

我们应该不难想起 20 世纪 80 年代时，数百家公司投身到“服务器-客户端”模式的浪

潮中,最后却发现,这种模式是如此复杂、应用起来又是如此的困难以至于软件的质量和可靠性下降到了无法使用的程度。

截至2009年,面向服务架构(SOA)模式占据了文献的大部分篇幅,并吸引了很多早期的信徒。但是我们是否可以预知SOA是一个真正成功的软件架构,还是仅仅在架构复杂性上一次毫无意义的飞跃?不幸的是,目前并没有足够的基于SOA的软件应用来帮助我们确信这种理论上非常有用的架构模式是否可以兑现它自己的承诺(请记住SOA应用并不是下载到用户的计算机中,而是远程运行在网络主机上,很显然,这种模式的效率取决于是否有足够的网络带宽和传输速度。如果网络上同时有数千个SOA应用和数百万个用户的时候,我们是否还有足够的网络带宽呢?没有人考虑过这个问题)。

另一种声称先进技术的架构模式是云计算。这种架构模式可以将软件应用进行分割,从而它们可以同时运行在数百台远程计算机上。考虑到这种模式下,云计算环境中的计算机可能包含非常差的安全协议,从而引发了人们对于这种架构下数据安全性的疑问。

概括来说,截至2009年,软件架构技术一直在飞速发展,因此对于企业来说,雇用专业的软件架构师是十分有价值的,这些软件架构师必须能够和软件架构风格的发展同步,从而始终掌握最前沿的技术。

但是为特定的软件应用选择合适的软件架构并不是一个非黑即白的单选题,负责这个软件应用的架构师需要结合所了解的软件架构设计原则以及所讨论软件的设计目的和功能等方面的信息做出恰当的选择。

7.6 企业架构师

在过去的30年中,随着计算机和软件应用越来越多地整合到企业运营当中,企业架构的重要性也日益增加。

在20世纪60年代,当大型主机第一次应用到商业领域时,这些计算机的性能在某种程度上是比较低的,同时应用范围也是比较有限的。因此,早期的商业应用更倾向于在本地运行,也就是说运行在某个特定数据中心内的指定计算机上,并且只为某个特定商业部门内的指定用户提供服务。

通常来讲一个企业都有很多个运营部门,包括制造部、市场部、销售部、财务部、人力资源部等,而大型企业还会同时有分布在不同城市的多个业务部门或制造部门。

当计算机和软件应用第一次正式成为商业工具时,一个通用的做法就是每个运营部门拥有自己的数据中心和自己开发的软件应用,通常情况下这些运营部门之间几乎不会针对这些软件的功能、界面或者相关数据进行沟通联络。

截至20世纪80年代,大型企业已经开发出了成百上千种商业软件,而其中大部分软件的服务范围都非常狭窄,而且仅仅面向本地用户。如果企业的管理者,比如说CEO,需要整合所有商业部门的信息以便制作全公司的商业报表,这个企业就需要从多个部门的软件系统中提取出数据,之后才能进行整合,这是一项非常耗时的工作,并且费用不菲。

这种棘手的情况导致了企业架构这一学科的出现,这门学科的主要目的是统一企业内

部多个运营部门之间的数据处理过程。同时，这种情况也催生了一个重要的商业软件市场：企业资源计划系统（Enterprise Resource Planning, ERP）。ERP 软件的基本概念是企业内部各个独立的商业软件很难被整合在一起，更经济有效的方式是使用一种大型的软件系统来代替它们。这种新的软件系统可以同时为企业内部所有运营部门提供服务，并且使用统一格式来存储所有这些部门的数据。

2000 年之后，世界上出现了数起企业欺诈案件和财务作弊案件，典型的例子就是安然公司^①。这些案例为企业架构又增添了一个新层面。企业架构师是公司内部软件治理的关键人物，他需要确保公司的财务数据是准确的，或者说确保公司员工能够对数据的准确性负责，否则该员工就要接受惩罚。

企业架构和之前讨论的软件架构的主要区别在于职责范围。通常情况下，软件架构只负责单一软件应用程序，其工作量范围大约从一万个功能点到超过十万个功能点，而企业架构师需要和企业内的软件组合打交道，其负责的软件规模大约在两百万个功能点到超过两千万个功能点。大型企业，如 IBM、微软或 Lockheed 公司，其企业内的软件组合通常包含数千个软件应用。

企业架构的另一个方面是，大型企业通常会开发和使用各种类型的软件：传统的信息管理软件、网络应用程序、嵌入式软件应用和系统软件。在这些软件中，一部分可能是内部员工开发的，一部分是外包人员开发的，一部分来自于商业供应商，一部分是开源软件，还有可能有一部分是通过合并或收购其他公司得到的。另外，在 2009 年的今天，任何大型企业的软件系统都必须要和其他公司的计算机系统进行连接，还要和一些政府机构进行连接，比如员工薪酬系统和税收系统。

企业架构中最困难的部分莫过于如何合并不同公司的软件组合，这通常是在合并或收购另一家公司之后要做的。通常情况下，各个公司软件组合所包含的软件应用中，有至少 80% 的软件应用在执行相似的功能，但是他们可能会使用不同的数据结构、不同的接口方式、不同的内部架构。

在合并公司之后将两个不同公司的软件组合整合为一个整体，这是所有企业架构师、软件架构师、业务分析人员甚至是所有软件工程专业人员都感到最为困难的一项工作。

企业架构师需要考虑的另外一个方面是，如何在完全不同的商业部门及其各自开发和维护的数据库内容之间进行通信。

除此之外，企业架构师还需要考虑许多技术方面的问题，这些问题包括但不限于硬件

① 安然公司曾是一家位于美国的得克萨斯州休斯敦市的能源类公司。在 2001 年宣告破产之前，安然拥有约 21 000 名雇员，是世界上最大的电力、天然气以及电信公司之一，2000 年披露的营业额达 1010 亿美元之巨。然而真正使安然公司在全世界声名大噪的，却是这个拥有上千亿资产的公司 2002 年在几周内破产，持续多年精心策划乃至制度化、系统化的财务造假丑闻。安然欧洲分公司于 2001 年 11 月 30 日申请破产，美国本部于 2 日后同样申请破产保护。从那时起，“安然”已经成为公司欺诈以及堕落的象征。——译者注

平台、软件操作系统、开源软件、外部供应商提供的 COTS 软件^①以及一些新兴话题比如云计算和面向服务架构,等等。尽管这些新兴的架构还没有被完全部署到企业内,但却是架构师们必须要考虑的一个方面。

企业架构与软件架构的关系正如同城市设计与建筑设计的关系。对于建筑设计师来说,他主要关心的只是一栋建筑,但是城市设计师需要同时考虑上千栋建筑。城市设计师必须要考虑城市所需要的基础设施以便满足不同功能区域的需求,这些功能区域包括居住区、商业区、工业区等。

表 7-8 向我们展示了企业架构的重要性如何随着企业内部软件应用数量的增长而增加。

表 7-8 向我们提出了一个非常有趣的问题:一个大型企业到底需要多少个企业架构师?由于企业架构师是一个全新的职业,因此这个问题并没有一个确切的答案。然而,考虑各种情况下的复杂性,一家企业应该为自己公司内部大约每 1000 个主要软件应用雇用企业架构师。因此,如果一家公司的软件组合中有 5000 个应用,那么这家公司大约需要 5 名企业架构师。

让我们换一种表达方式,每个企业架构师能够负责的工作范围大约从 50 万个功能点到超过 200 万个功能点。

对于一家大型企业如 IBM、微软或 Unisys 公司^②来说,其内部完整的软件产品组合可能包括:

- 3000 个内部信息管理软件
- 1500 个网络软件应用
- 1000 个工具软件(项目管理工具、测试工具等)
- 3500 个来自其他公司的商业软件(ERP 软件、人力资源软件等)
- 2000 个卖给其他公司的商业软件
- 2500 个系统软件
- 500 个嵌入式软件应用(安全应用、自动控制系统(AC)等)

表 7-8 企业架构的价值随企业内部软件数量的变化

企业内部软件应用数量	企业架构的重要性
10	不需要企业架构
100	企业架构很有用
1 000	企业架构很重要
10 000	企业架构非常重要
100 000	企业架构具有决定性作用
1 000 000	企业架构具有决定性作用但架构设计通常很困难

① COTS 软件,即商务现货供应软件,是指使用“不再做修理或改进”的模式出售的商务产品,这种产品设计的原则就是安装简便,并且可以在现有系统部件的条件下运行。通常电脑用户所要买的软件几乎都可以是 COTS 类别的产品:操作系统、Office 产品组合、字处理以及电子邮件程序就是其中的几个例子。COTS 软件的最大优点就是它的大量生产以及相对的低成本。——译者注

② Unisys 公司是一家闻名全球的信息技术服务及解决方案提供商,为遍布世界 100 多个国家和地区的客户提供先进的技术,帮助他们在全球经济中捕捉商机,迎接挑战,获得成功。Unisys 公司提供企业、政府在转型过程中所需的解决方案、服务、平台和网络基础设施。——译者注

250 个开源软件

总计 14 250 个软件应用

如果我们假设这个数量是公司内部软件应用的总数量，那么这家公司大约需要雇用 15 名企业架构师。

这些完全不同的软件应用可能运行在数十种硬件平台上和若干种操作系统上。换句话说，一家大型公司的软件世界就如同一家丰富多彩的自助餐厅，里面陈列了多种多样的软件应用、软件硬件平台、数据文件结构、通信接口以及其他容易出现问题的领域的产品。

截至 2009 年，在面向服务架构（SOA）、云计算的出现及开源软件爆发式发展的影响下，企业架构师的地位有了很大的发展，这同时也得益于《萨班斯-奥克斯利法案》强制要求企业提供更加准确的财务报告的新标准。

全球经济的衰退也将会对企业架构产生重大影响，但究竟是何种影响暂时还无法预计。在一个企业大量裁员、业务部门被迫关闭、未完成的软件应用停止开发以及开发人员和维护人员大量减少的时代，我们并没有一个模型或者指导纲领可以知道什么将会发生在企业架构师们身上。事实上，企业架构师有可能也是被裁掉员工中的一部分，因为人们并不总是认为他们的工作会对公司的基本运营产生直接影响。

在企业架构领域有一些非营利性组织可以对企业架构师们提供帮助，其中之一就是企业架构师协会（Association of Enterprise Architects, AEA），该协会的网址是 aeaassociation.org。

另外一家协会有着非常拗口的名字，叫做开放集团企业架构师协会（Association of Open Group Enterprise Architects, AOGEEA），其网站是 AOGEEA.org。这家协会是由开放集团组织（Open Group Organization）和全球企业架构组织（Global Enterprise Architecture Organization）合并而成，其拗口的名字也是由两家的名字合在一起组成的。合并后，这家组织声称自己是全球最大的企业架构组织。

同时，世界上还有一家关于企业架构的期刊，叫做企业架构期刊（The Journal of Enterprise Architecture, JEA），由企业架构师协会出版。

我们通常很难找到关于企业架构师为企业所做的特定计划，因为他们的工作通常专属于某个企业而不对外公开。但是，美国联邦政府的许多部门及大多数州政府确实对外界公开了他们企业架构的一些信息。美国国防部，作为世界上最大的软件用户，正在试图开发出一种新的更加优秀的企业架构。

很明显，企业架构师对于黑客技术、蠕虫、病毒及拒绝服务攻击（Denial of Service Attack）的巨大发展应该密切关注并重点考虑。然而，安全性能的提高需要专门技能，而这些技能在今天非常缺乏，因此通常人们会找到一些外部安全专家来进行咨询，以帮助企业架构师进行工作，直到他们掌握了这些安全性能方面的技能。

从最佳实践的角度来说，拥有超过 500 个软件应用的组织应该雇用至少一名企业架构师。而拥有超过 5000 个软件应用的大型公司可能需要 5 名企业架构师，正像之前讨论的软件应用和企业架构师之间的比例。

各个公司的企业架构师的具体职责可能相差很多，因此很难去定义一些最佳实践。很

明显,在公司部门间提高数据共享幅度应该算做一个最佳实践。减少多余的软件应用、减少过时的软件应用和笨重的遗留应用应该算做另一个最佳实践。而其他的一些做法,可能会被视作最佳实践,也可能不会,这些做法包括减少 COTS 软件的比例而将其改为内部软件,参与到企业内部 ERP 软件的选择和部署工作中,等等。毫无疑问,企业架构师的工作仍将会随着技术和业务的改变而改变。

7.7 软件设计

假设你所在公司的 CEO 请你对最近公司开发的 250 个内部软件进行一次检查,以便找出建立可重用组件库的备选功能组件,这些组件包括设计、代码和测试用例。你该如何完成这项任务呢?

以 2009 年左右的技术水平来说,这项任务并不容易完成。在这 250 个软件应用中,大约会有 75 个是少于 1000 个功能点的小型软件,它们很可能会使用敏捷开发方式而且使用用户故事作为主要的设计描述方式,同时也可能会混合使用其他的描述方法。对于单个的软件应用来说,用户故事是非常有用的,但如果需要找出多个软件应用中的共同点,用户故事就显得不是那么有效了。

还可能会有大约 50 个软件是超过 5000 个功能点的大型商业软件,这些软件很可能是用了多种正式的设计描述方式,也或许会使用 UML 方法来描述从联合应用设计(JAD)方法中收集到的需求。尽管 UML 方式可以帮助我们为单独的软件应用建立模型,但是考虑到如此多的各具特点的 UML 图表,如果我们要想通过审视大量项目的 UML 图表(如 50 个项目)来试图找出其中共有的功能,这仍然不是一件容易的或者很快就可以完成的工作。

自动化的工具,例如静态分析工具,也许可以通过分析基于 UML 的元语言的语法结构来找出共有的模型,但在 2009 年左右,这项技术还不能应用到实践中。

在这 250 个软件应用中,还可能会有 25 个是科研项目软件或工程项目软件,它们可能会使用状态变化图、建模语言(如 LePus3 语言^①、Express 语言^②)或者质量功能展开(QFD)方法所建立的“质量屋”图表以及其他多种架构建模元语言。

余下的 100 个软件应用可能使用了多种描述方法,包括但不限于用例、UML 方法、N-S 图、Jackson Design、流程图、决策表、数据流向图、HIPO 图以及其他各种方式。其中的一些方法可能会定义模型,但即使是对 100 个项目进行扫描检查也不是一件容易的事情。

总结来说,这 250 个最新开发的软件应用使用了超过 50 种不同的设计语言和方法,而对其中的大部分语言和方法来说,进行相互转化是一件非常困难的工作。同时,这些语言和方法也很难通过自动化验证工具和自动化错误检查工具来处理。

① LePus3 语言,即 Language for Pattern Uniform Specification 语言的简称,是一种针对面向对象语言编程和设计的可视化建模语言。——译者注

② EXPRESS 信息建模语言是产品建模数据交换标准 STEP 和使用 STEP 的产品数据交换规范 PDES (product data exchange specification) 中的产品信息建模语言。它提供了一种以形式化的方式精确定义产品数据的手段。——译者注

通过以上分析我们可以看到，在同一个公司内的 250 个软件应用样例使用了如此多完全不相容的设计描述方式，因此我们几乎不可能通过设计文档来找出多个软件应用中共同的功能或者模型。这就意味着，如果我们想要为可重用组件库挑选一些备选组件，会是一项非常困难的工作。

既然所有软件应用都已经开发完成并且在运行当中，假设所有这些软件都是使用 C、C++、Java 等大约 25 种静态分析可以应用的编程语言开发的，那么或许我们可以通过对软件源代码进行静态分析来得到这些软件的共用模型。

由于所有的设计描述方式都是基于元语言的，使用静态分析工具来分析设计文档在理论上是可行的，但是截至 2009 年，静态分析工具应用的对象还只是编程语言而不是元语言。

我们还可以使用遗留应用的修复工具来寻找公用模型，这些修复工具通过对遗留应用的源代码进行分析从而用一种更易于维护和修复的方式来显示代码。通过使用一种或多种修复工具从而找出公用模型也是有可能实现的。

还有一种可能就是使用一些更加复杂的复杂度分析工具，这些工具会分析软件源代码从而分析出软件的基本复杂度和循环复杂度，同时也会分析出源代码中的模型。

概括起来说，截至 2009 年，从软件代码中找到公用模型比从设计文档中定义模型要容易，但是这并不是一个正常的现象。自动化分析工具应该可以对设计方式进行处理以便发现缺陷和定义可重用组件的模型。

软件设计的另外一个问题是，如果将所有软件缺陷按照阶段分类（即需求缺陷、设计缺陷、代码缺陷等）并按照数量多少进行排名的话，设计缺陷是排名第二的缺陷种类。平均下来，每个功能点大约包含 1.25 个设计缺陷或错误，而同时在软件代码中，每个功能点大约包含 1.75 个缺陷。

在考虑到每个功能点的设计文档大约是一页到两页之间，那么上面数据的言外之意就是说，基本上设计说明书的每一页都至少包含一个缺陷或者错误。这就是设计审查对于减少软件设计问题是如此强大和有效的原因。

考虑到无论设计描述是否使用了用例、UML、流程图或者其他约 50 种描述方法中的一种或多种，软件设计中典型错误的比例仍然很高，因此我们并没有足够证据来说明其中任何一种设计方法可以作为最佳实践。或许，更加有用的工作是去考虑哪些基础性的话题应该加入到软件设计中，作为设计工作的一部分。

软件设计视图

客观来说，软件设计应该算做另一个更加一般性的话题——知识展示的一个子集。这就向我们提出了另外一些非常重要的问题，首先，当设计一个软件应用时，什么样的内容是应该展示的；其次，对于软件设计需要考虑的众多方面来说，什么样的展示形式或者选择那种语言进行展示才是最好的。

由于在设计阶段，软件产品还没有成为一个可视化的实体并且有很多动态特征，因而我们很难去列举需要展示软件产品的哪些方面。和一些静态物体（如建筑物）的设计相比，软件设计因其动态特征，因此要决定展示的内容就更加困难。以下 8 个概述性问题是我们的

在进行软件特征展示的时候需要讨论的：

1. **外部视图**，用来展示从明确的用户需求产生的用户可见的功能特性。外部视图包括展示给用户的屏幕图像、报告格式以及用户对于嵌入式系统进行操作之后的回应。这个视图可能会定义一些可以和其他软件应用共享的功能特性，这部分功能就是可重用的组件。同时，这个视图还需要包含其他一些内容：

对于用户误操作的容错处理程序

软件应用运行的硬件平台和软件平台

软件应用支持的国家/地区以及语言

这个视图的描述应该非常简练，平均起来，每个功能点的描述大约占半页到一页的篇幅。

2. **运算法则视图**，用来展示软件应用中包含的数学公式及逻辑运算。这些公式和运算可能是非常直接的运算，如货币转换，也有可能是非常复杂的，如量子力学中的运算公式。无论哪种情况，在对这些公式和运算进行编码前，我们需要对软件应用中包含的主要运算法则进行展示并加以解释。这个视图的描述应该非常简练，平均下来每个功能点的描述大约占四分之一页的篇幅。

3. **结构视图**，用来展示软件应用中包含哪些组件和模块，以及这些组件和模块如何连接以构成一个整体的软件应用。这个视图需要包括：

这些模块按照何种顺序运行

哪些模块是并发运行的

外部应用对该软件应用的调用接口

从外部应用或者定制的某特定应用中重用的模块或功能

软件应用使用的面向对象语言中的类或继承

这个视图是所有视图中最复杂冗长的一个，每个功能点的描述大约要占据一到两页的篇幅。

4. **数据视图**，用来展示该软件应用将会创建、使用和操作何种数据信息。这个视图会包含关于数据的一些信息，比如数据是否是由商业信息、符号、基于传感器的信息、图像、声音或其他一些内容组成。举例来说，植入耳蜗的假耳中内置的软件需要将外部的声音信号转换为电信号。由于截至2009年，软件工业还没有一个“数据点”的度量准则或者其他度量方法来表达数据库、数据存储或数据仓库的规模，因此我们还无法找到一个有效的方式来描述软件将要使用的数据规模或容量。

5. **属性视图**，用来展示软件部署后的一些非功能性目标或指标。这些属性可以包括以下几点：性能（用运行速度来衡量）、可靠性（用平均无故障时间MTTF来衡量）以及其他的一些属性参数。这个视图也是非常简洁的，无论该软件应用的规模大小，这个视图通常都不会占据超过三页的篇幅。

6. **安全性视图**，用来展示在遇到病毒、蠕虫、搜索机器人、拒绝服务攻击以及其他攻击方式的时候，软件应用如何保护自己免受侵害。其他攻击方式包括各种试图干涉软件正常运行或试图偷取软件内信息的行为。这个视图是大约在2009年新出现的，但它很快成为了软件应用设计中的一个标准配置，特别是对于财务软件、健康卫生业软件以及其他任何

包含有价值信息或机密信息的软件应用。截至 2009 年,这个新出现的视图尚没有任何对其进行规模估算的信息,但很有可能这个视图也是非常简洁的。

7. **模式视图**,或者说是其他在多个软件应用中可能出现的视图的综合体。正因为它在多个软件应用中都可能出现,因此它所包含的信息应该是可重用的。典型的有可重用潜能的模型包括各个软件中相似的外部功能、相似的运算法则以及相似的数据结构。软件模型还可能包括面向对象软件中的类和继承。在这个视图中,根据所要描述的可重用组件自身的规模大小,描述某个特定的模型大约需要 0.1 到 0.4 页的篇幅。

8. **后勤视图**,用来记录软件某些特定的历史数据,通常是容易丢失或者很难找寻的数据。这些历史性的数据通常包括软件第一次投入使用的日期、地点、软件开发过程涉及的公司、软件开发过程中使用的方法、工具及实践经验。在后勤视图中,还应该包括软件的规模,这方面应该从功能点角度和逻辑代码行的角度同时提供数据,同时还应该包括软件开发过程中使用的所有编程语言。由于软件规模会随着软件的使用不断增长,后勤视图还应该记录需求蔓延的情况以及在后续的所有版本中软件规模的增长情况。后勤视图还要包括软件中可重用材料的来源。这个视图是为了协助建立标准参照系统而设计的,同时还对多种回归分析有帮助,这些回归分析是为了证明一些开放方式如敏捷开发方法或 TSP 方法的有效性而进行的。该视图中的部分内容可以作为在一些正式的软件分类系统中对该软件应用进行记录的信息,比如我们本章之前讨论的分类系统。无论软件应用自身规模大小,这个视图通常不会超过 10 页。

当以上的 8 个视图信息合并到一起后,设计文档的平均规模大约是每个功能点 3 页文档,而每个功能点的文档数量范围可以从不到 1.5 页到超过 6 页。

由此我们可以看到,软件设计需要进行相当大量的文档开发工作。因此我们便很容易可以理解为什么大型软件项目中开发文档的费用比开发代码的费用还要高,也会容易理解为什么一些敏捷开发概念对于软件设计实践中的大量文档和巨额开销持保守态度。

考虑到在软件设计阶段要提供如此之多的视图文档,很明显任何一种语言或描述方式都不能完成所有 8 种视图文档。因此,在设计过程中我们必须利用多种方式来展现设计内容:

- ❑ 自然语言可以用来定义属性视图、后勤视图以及外部视图图中的一些内容。一些特殊形式的自然语言,如可执行语言,也可以使用。
- ❑ 外部视图图中的一些内容可以使用图像来展示,如一些代表性的屏幕截图或者输出示例。
- ❑ 运算法则视图需要使用数学公式或者其他形式的科学符号。
- ❑ 结构视图需要使用一些图表或标志符号。由于软件的动态属性,如果能够使用一些动画的形式来进行展现,那将会比静态视图更加受欢迎和易于理解。同时,通过进行动画模拟,我们也可以在设计时为软件的性能建立模型。

由于对多种设计描述方式使用自动化验证有一定困难,如果我们可以将软件的主要视图转换为同一种元语言,那么这项工作将是非常有价值的,也会产生很大的帮助。很明显,大部分视图的信息都会转化为软件代码,但如果等到软件代码开发完成的时候再做设计验

证和检验已经太迟了。

无论一种被广泛接受的元语言是基于某种形式的巴科斯范式^①或是某种定子句文法(DCG)又或者是其他什么,它都应该包含一些可以进行自动化分析的属性,从而达到可以进行自动化验证和检验的目的。在此之后,我们可以试图再进一步,通过分析元语言来产生一系列的测试用例,这也是可能实现的。

概括来说,2009年左右,软件设计师可以使用大约50种描述方法来展现设计内容,这些方法对于单个软件应用来讲都是有效的,但是如果我们想要对多个软件应用进行分析以定义公用模型和找到适合成为可重用组件的功能,那么这50种方法中没有任何一种是有效的。

7.8 总结

长久以来,在软件应用进行代码开发前进行各种文档的开发一直是软件工程领域一件让人头疼的工作。这些文档需要对所开发的软件进行描述,但各种形式的文档中都会发现很多错误和缺陷。同时,在从需求到设计的转化过程中,以及从设计到代码的转化过程中,我们总会有意无意地漏掉某些功能需求,而且总有一些不在需求中的功能被添加进来。

通常情况下,开发这些文档的费用甚至比开发产品代码的费用还要高。尽管我们可以通过审查的方式来发现文档中的错误,而且文档审查也是一种非常有效的方式,但是对文档进行自动化审查仍然是一件很困难的事情,无论这些文档是文字设计还是图形设计。

在软件产品所有的缺陷中,有大约60%是在需求收集、分析、架构及设计过程中出现的,而这些过程的开销大约占总项目费用的30%到40%。实际上,对于大型软件应用开发来说,占据费用前三位的项目分别是:

1. 发现和修复缺陷(这些缺陷很多源自文档)
2. 开发各种文档,包括需求文档、架构文档以及设计文档
3. 开发产品代码

由于文档中所包含的缺陷比产品代码本身所包含的缺陷要多,同时文档开发费用也比代码开发费用要高,因此软件工程行业的研究人员们需要继续进行研究,一方面要减少各种文档中的缺陷,另一方面要降低文档开发工作的费用。

希望未来的研究可以帮助我们更加容易地发现软件模型,同时可以帮助我们更加有效地对需求和设计进行自动化验证。

截至2009年,对于需求文档、架构文档和设计文档进行正式审查,是减少这些重要文档中的缺陷的最有效方法。但是这种审查通常很耗时,并且费用较高。由于静态分析和测试方法都不足以胜任减少需求和设计文档缺陷的重任,因此手工审查仍然是目前最主要的方式。

^① Backus-Naur Notation, 又可写作 Backus-Naur Form, BNF, 是由 John Backus 和 Peter Naur 首次引入一种形式化符号来描述给定语言的语法(最早用于描述 ALGOL 60 编程语言)。现在,几乎每一位新编程语言书籍的作者都使用巴科斯范式来定义编程语言的语法规则。——译者注

参考文献

注：世界上有超过 500 本书籍及数千篇期刊文章针对软件需求、商业分析、软件架构、企业架构及软件设计进行了讨论。但尽管有如此多的出版物，这些领域的工作在软件工程行业中仍然是令人头疼的，并且经常出现错误。下面列出的资料仅仅是这些文献中的一部分示例。

软件项目文档的质量及费用

Beck, Kent. *Test-Driven Development*. Boston, MA: Addison Wesley, 2002.

Cohen, Lou. *Quality Function Deployment—How to Make QFD Work for You*. Upper Saddle River, NJ: Prentice Hall, 1995.

Cohn, Mike. *Agile Estimating and Planning*. Englewood Cliffs, NJ: Prentice Hall PTR, 2005. (中文版《敏捷估计与规划》，宋锐译，清华大学出版社 2007 年 7 月出版)

Garmus, David and David Herron. *Function Point Analysis—Measurement Practices for Successful Software Projects*. Boston, MA: Addison Wesley Longman, 2001. (中文版《功能点分析——成功软件项目的测量实践》，钱岭、苏薇、盛铁阳译，清华大学出版社 2003 年 12 月出版)

Garmus, David and David Herron. *Measuring the Software Process: A Practical Guide to Functional Measurement*. Englewood Cliffs, NJ: Prentice Hall, 1995.

Gilb, Tom and Dorothy Graham. *Software Inspections*. Reading, MA: Addison Wesley, 1993.

Glass, R.L. *Software Runaways: Lessons Learned from Massive Software Project Failures*. Englewood Cliffs, NJ: Prentice Hall, 1998. (中文版《软件开发的滑铁卢：重大失控项目的经验与教训》，陈河南等译，电子工业出版社 2002 年 2 月出版)

Harris, Michael, David Herron, and Stacia Iwanicki. *The Business Value of IT: Managing Risks, Optimizing Performance, and Measuring Results*. Boca Raton, FL: CRC Press (Auerbach), 2008.

Humphrey, Watts. *Managing the Software Process*. Reading, MA: Addison Wesley, 1989. (中文版《软件过程管理》，高书敏、顾铁成、胡寅译，清华大学出版社 2003 年 3 月出版)

Jones, Capers. *Assessment and Control of Software Risks*. Englewood Cliffs, NJ: Prentice Hall, 1994.

Jones, Capers. *Estimating Software Costs*. New York, NY: McGraw-Hill, 2007. (中文版《软件项目估计》，刘从越、郝建材、申冬凯译，电子工业出版社 2008 年 3 月出版)

Jones, Capers. *Patterns of Software System Failure and Success*. Boston, MA: International Thomson Computer Press, 1995.

Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Boston, MA: Addison Wesley Longman, 2000. (中文版《软件评估：基准测试与最佳实践》，韩柯等译，机械工业出版社 2003 年 4 月出版)

Jones, Capers. "Software Project Management Practices: Failure Versus Success." *CrossTalk*, Vol. 19, No. 6 (June 2006): 4–8.

Jones, Capers. "Why Flawed Software Projects are not Cancelled in Time." *Cutter IT Journal*, Vol. 10, No. 12 (December 2003): 12–17.

Kan, Stephen H. *Metrics and Models in Software Quality Engineering*, Second Edition. Boston, MA: Addison Wesley Longman, 2003. (中文版《软件质量工程——度量与模型》(第二版)，吴明晖、应晶译，电子

工业出版社 2004 年 7 月出版)

- McConnell, Steve. *Software Estimating: Demystifying the Black Art*. Redmond, WA: Microsoft Press, 2006.
- Radice, Ronald A. *High Quality Low Cost Software Inspections*. Andover, MA: Paradoxicon Publishing, 2002.
- Roetzheim, William H., and Reyna A. Beasley. *Best Practices in Software Cost and Schedule Estimation*. Upper Saddle River, NJ: Prentice Hall PTR, 1998.
- Strassmann, Paul. *Governance of Information Management: The Concept of an Information Constitution*, Second Edition. (eBook). Stamford, CT: Information Economics Press, 2004.
- Strassmann, Paul. *Information Payoff*. Stamford, CT: Information Economics Press, 1985.
- Strassmann, Paul. *Information Productivity*. Stamford, CT: Information Economics Press, 1999.
- Strassmann, Paul. *The Squandered Computer*. Stamford, CT: Information Economics Press, 1997.
- Wiegiers, Karl E. *Peer Reviews in Software—A Practical Guide*. Boston: Addison Wesley Longman, 2002.
- Yourdon, Ed. *Death March—The Complete Software Developer's Guide to Surviving "Mission Impossible"* Projects. Upper Saddle River, NJ: Prentice Hall PTR, 1997. (中文版《死亡之旅》, 周浩宇、杨华译, 机械工业出版社 2012 年 1 月出版)

软件需求

- Artow, J. & I. Neustadt. *UML and the Unified Process*. Boston: Addison Wesley, 2000.
- Booch, Grady, Ivar Jacobsen, and James Rumbaugh. *The Unified Modeling Language User Guide*, Second Edition. Boston: Addison Wesley, 2005.
- Cockburn, Alistair. *Writing Effective Use Cases*. Boston: Addison Wesley, 2000. (中文版《编写有效用例》, 王雷、张莉译, 机械工业出版社 2002 年 9 月出版)
- Cohn, Mike. *User Stories Applied: For Agile Software Development*. Boston: Addison Wesley, 2004.
- Fernandini, Patricia L. *A Requirements Pattern. Succeeding in the Internet Economy*. Boston: Addison Wesley, 2002.
- Gottesidner, Ellen. *The Software Requirements Memory Jogger*. Salem, NH: Goal QPC Inc., 2005.
- Inmon, William H., John Zachman, and Jonathan G. Geiger. *Data Stores, Data Warehousing and the Zachman Framework*. New York: McGraw-Hill, 1997.
- Orr, Ken. *Structured Requirements Definition*. Topeka, KS: Ken Orr and Associates, Inc., 1981.
- Robertson, Suzanne and James Robertson. *Mastering the Requirements Process*, Second Edition. Boston: Addison Wesley, 2006. (中文版《掌握需求过程》(第二版), 王海鹏译, 人民邮电出版社 2007 年 6 月出版)
- Wiegiers, Karl E. *Software Requirements*, Second Edition. Bellevue, WA: Microsoft Press, 2003. (中文版《软件需求》(第二版), 刘伟琴、刘洪涛译, 清华大学出版社 2004 年 11 月出版)
- Wiegiers, Karl E. *More About Software Requirements: Thorny Issues and Practical Advice*. Bellevue, WA: Microsoft Press, 2000.

商业分析

- Carkenord, Barbara A. *Seven Steps to Mastering Business Analysis*. Ft. Lauderdale, FL: J. Ross Publishing, 2008. (中文版《七步掌握业务分析》, 朱庆、蒋慧、甄进明译, 电子工业出版社 2010 年 9 月出版)

Haas, Kathleen B. *Getting it Right: Business Requirements Analysis Tools and Techniques*. Vienna, VA: Management Concepts. 2007.

软件架构

- Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Boston: Addison Wesley, 1997. (中文版《软件构架实践》(第二版), 车立红译, 清华大学出版社 2004 年 3 月出版)
- Marks, Eric and Michael Bell. *Service-Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology*. New York: John Wiley & Sons, 2006.
- Reekie, John and Rohan McAdam. *A Software Architecture Primer*. Angophora Press, 2006.
- Shaw, Mary and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice Hall, 1996.
- Taylor, R.N., N. Medvidovic, E.M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Hoboken, NJ: Wiley, 2009.
- Warnier, Jean-Dominique. *Logical Construction of Systems*. London: Van Nostrand Reinhold, 1978.

企业架构

- Bernard, Scott. *An Introduction to Enterprise Architecture*, Second Edition. Philadelphia, PA: Auerbach Publications, 2008.
- Fowler, Martin. *Patterns of Enterprise Application Architecture*. Boston, MA: Addison Wesley, 2007. (中文版《企业应用架构模式》, 王怀民、周斌译, 机械工业出版社 2010 年 4 月出版)
- Lankhorst, Marc. *Enterprise Architecture at Work: Modeling, Communication, and Analysis*. Cologne, DE: Springer, 2005.
- Spewak, Steven H. *Enterprise Architecture Planning: Developing a Blueprint for Data, Applications, and Technology*. Hoboken, NJ: Wiley, 1993.

软件设计

- Ambler, S. *Process Patterns—Building Large-Scale Systems Using Object Technology*. Cambridge University Press, SIGS Books, 1998.
- Berger, Arnold S. *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*. Burlington, MA: CMP Books, 2001.
- Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Design*. Boston: Addison Wesley, 1995. (中文版《设计模式: 可复用面向对象软件的基础》, 李英军、马晓星、蔡敏、刘建中译, 机械工业出版社 2000 年 6 月出版)
- Martin, James & Carma McClure. *Diagramming Techniques for Analysts and Programmers*. Englewood Cliffs, NJ: Prentice Hall, 1985.
- Shalloway, Alan & James Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Second Edition. Boston, MA: Addison Wesley Professional, 2004. (中文版《设计模式解析》(第二版), 徐言声译, 人民邮电出版社 2006 年 10 月出版)

编程和代码开发

8.1 引言

在本章中，笔者提出了一些在其他软件工程书籍中不太常见的观点。除了一些常见话题之外，本章还讨论了以下 12 个其他软件工程文献中不经常讨论的问题：

1. 我们为什么会有超过 2500 种编程语言？
2. 尽管我们已经有如此多的编程语言，为什么平均每个月仍然有超过一种新编程语言出现？
3. 对于软件工程来说，我们到底需要多少种编程语言？
4. 为什么现今大多数软件应用都使用 2 ~ 15 种不同的编程语言？
5. 在现今仍然在维护使用的软件应用中，有多少是使用已经过时了、已经极少有人使用的编程语言编写的？
6. 有多少程序员在使用主流编程语言？又有多少程序员在使用小众编程语言？
7. 我们是否应该有一个国家性质的翻译中心来完成以下工作？
 - 维护已经过时的编程语言的编译器和工具
 - 将古老的已经几乎无人使用的编程语言转化为现代编程语言
8. 在软件源代码中发现的主要缺陷类型有哪些？
9. 与审查相比，软件测试工具和静态分析工具的作用如何？
10. 在缺陷去除方面，各种类型的测试方法的有效性有多高？
11. 可重用的代码在质量、安全和开发费用方面有多大作用？
12. 在软件经济学研究中，“代码行”的度量标准为什么不再有效？

软件编程中的重要话题并不仅仅是以上 12 个，但是其他的软件工程书籍和期刊很少涉及对它们的讨论。以下就是我们对于这 12 个问题的讨论。

8.2 编程语言开发简史

如果我们回顾一下计算机程序和编程语言开发的历史，会发现很多有意思的事情。早期由齿轮和传动装置驱动的机械计算机，及后来使用打孔卡的计算机的历史很有意思，但

是它们和计算机程序并不相关。尽管这些装置确实表现出了计算机程序的一些本质性含义，那就是通过千变万化的各种指令来控制机械装置的动作已达到改变其所执行动作的目的。

计算机领域的先驱者们包括 Charles Babbage、Ada Lovelace、Hermann Hollerith、Alan Turing、John Von Neumann、Conrad Zuse、J. Presper Eckert、John Mauchly，等等。John Backus、Konrad Zuse 等人的工作为编程语言的建立奠定了基础。David Parnas 和 Edsger Dijkstra 为结构化编程语言的开发做出了贡献，结构化编程帮助人们将代码中错综复杂的分支减少到了最小。在此之前，程序代码中的分支路径越来越多而形成了“意大利面条碗”^①以至于人们根本无法阅读。

Ada Lovelace 是 Charles Babbage 先生的助手，在 1842 ~ 1843 年间，她设计了一种方法来在巴贝奇分析机上面对于伯努利数列的计算。她的这项工作被认为是世界上最早的计算机程序，尽管有些人对此仍有争议。

在第二次世界大战前期及战争期间，许多国家的公司建造了基于机电系统的计算装置，主要是为了一些特殊用途，如计算弹道、完成数学计算任务等。

最早的“编程”模型是通过改变不同线路之间的连接或者使用插接板来完成的。但在二次世界大战期间，人们开发出了拥有存储设备的计算装置，这些存储装置可以储存数据和指令。能够通过存储装置存储指令是计算机程序发展的一大进步，这一进步为我们打开了现代计算机编程的大门。

德国科学家 Konard Zuse 在 1941 年建造了 Z3 计算机，之后在 1948 年设计出了似乎是世界上最早的高级语言——Plankalkul 语言，尽管这种语言并没有自己的编译器，甚至没有人使用这种语言进行程序开发。

存储在计算机中的早期“语言”主要是二进制代码和机器语言。显然人们很难去理解这些语言，更不要提使用这些语言进行编程或者修改已有的程序。这种困难直接导致了编程语言的诞生，人们可以理解编程语言并使用它们进行操作，同时也可以翻译成为机器指令。

最早的编程语言叫做“汇编语言”，通常情况下，汇编语言对于人类可以理解的指令（称为源代码）和可执行的机器指令（称为目标代码）有一个一一对应的映射关系。

于是，人们发现了一种新的做法，那就是通过开发一种语言，使得人类可以使用这种语言描述各种数据计算法则或者其他数据操作方式。人们很快发现这种做法非常有效，于是便开发出了更多种专门领域的语言。

这些语言的语法为人类解决某种特定的问题做了优化处理以便人类能够更好地理解，而将这些语言翻译成为机器代码的工作则留给了编译器来完成。顺便提一下，汇编程序和编译器的主要区别在于，汇编程序更倾向于使用一一对应的源代码和目标代码，而编译器使用的是一对多的关系。换句话说，编译器中的一行代码可能会生成十几条甚至更多的机

① “意大利面条碗”现象（Spaghetti bowl）一词源于巴格沃蒂（Bhagwati）1995 年出版的《美国贸易政策》（U.S. Trade Policy）一书，指在双边自由贸易协定（FTA）和区域贸易协定（RTA），统称特惠贸易协议下，各个协议的不同优惠待遇和原产地规则。原产地规则就像碗里的意大利面条，一根根地绞在一起，剪不断，理还乱。这种现象贸易专家们称为“意大利面条碗”现象或效应。此处借用这个词指代软件代码中的分支错综复杂，就像意大利面条一样绞在一起，无法阅读。——译者注

器指令。

从将一条单独的源指令转化为许多条目的指令的能力产生了高级编程语言的概念。总体来说,编程语言越高级,从一行源代码中转换出来的目标代码就越多。

无论是汇编程序还是编译器,都是由特殊的转换程序通过批处理来实现的。由于转换程序的运行需要时间,这些程序的源代码并不能在编写完成之后立即执行,有的时候由于计算机在处理其他工作而其他计算机又不可用,转换工作可能需要延迟数小时才能进行。这种延迟现象使得人们开始去开发其他的代码转换方法,于是一种新的编程语言转换器很快出现并命名为“翻译器”。翻译器可以实时地将源代码转换为目标代码。

在计算机和程序开发的早期,人们使用软件的范围非常小,主要用来解决数学计算领域的问题。但是随着数字计算机运行速度的提高,软件的应用范围也越来越广。当人们开始将计算机的应用扩展到商业领域并开始操作文本和数据的时候,如果源代码中能够包含一些所应用领域特有的词汇和语言,那么很明显人们会更加容易地学习和使用这种语言。而人们开始使用计算机来控制物理装置也引出了另外一个需求,那就是编程语言需要针对物理装置做进一步优化。

于是,人们开发了大量针对专门领域的编程语言,其目的是用于专门的领域,比如表处理、商业应用、天文学应用、嵌入式软件,等等。

8.3 我们为什么会有超过 2500 种编程语言

软件程序源代码需要对特定种类的商业问题或技术问题进行处理,这一概念是导致编程语言数量大幅增加的原因之一。

编程语言使用的词汇和个别应用领域的术语保持一致有一些技术方面的好处,比如说,对于这个特定领域非常熟悉的编程人员很容易就可以学会使用这门语言。

实际上,开发一门新的编程语言是一件非常容易的事情。随着越来越多的领域开始使用计算机解决问题,越来越多的编程语言也出现了。如果能够成功开发出一门语言并吸引其他开发人员来使用,那么开发者也会获得一定的社会声望。

正是由于上述技术原因和社会原因,软件行业中新编程语言以惊人的速度增加着。截至 2009 年的今天,没有人能够了解现今存在的编程语言及各种派生语言的确切数字,但现今出版的最大的编程语言列表(由 Bill Kinnersley 出版的语言列表, <http://people.ku.edu>)中包含了超过 2500 种语言。

笔者之前所在的公司,美国软件生产力研究所(Software Productivity Research, SPR)自 1984 年以来一直保存着一份常用编程语言的列表,而这份列表的最新版本包含了 700 多种语言。新的编程语言以每月两到三种的速度持续出现,有些月份甚至出现了超过 10 种,而这种增长目前还无法看到尽头。

产生了如此之多的编程语言的原因之一便是一个软件工程师仅用一到两个月的时间里就可以开发出一门新的语言。事实上,借助编译器编译程序(Compiler-compiler,是一个帮助使用者根据某种语言或机器的规则来产生语法分析器或者编译器的工具),一门新的编程

语言从头脑中一个模糊的概念到成为编译好的代码，只需要 60 天甚至更少的时间。

1984 年，笔者的第一款商业软件估算工具投入市场。该工具的初始版本提供了估算软件开发费用及质量的功能并支持 30 种不同的开发语言，但是这个工具本身也可以应用在其他开发语言上面，其工作的逻辑和运算法则都是相同的。因此，我们对客户做了如下声明：我们的工具可以支持“所有已知的开发语言”的费用估算。

既然有了这样一个声明，我们就有必要收集一个包含所有已知编程语言及其版本的列表作为备用。在 1984 年我们做出如上声明的时候，笔者预计这个列表大约会包含 50 种语言，然而，当数据收集工作结束的时候，我们发现，在 1984 年，这个“所有已知编程语言”列表包含了大约 250 种语言及其派生语言。

在收集这份语言列表的同时，我们还发现了另一个现象，那就是每个月都会有一些新的编程语言出现，有时候甚至有很多种。很明显，想要了解所有编程语言的最新动态绝对不是一件可以快速和轻易完成的事情，而是需要投入持续不断的精力。

在 2009 年的今天，这份由美国软件生产力研究所进行维护的列表的最新版本已经包含了超过 700 种编程语言，而新编程语言的出现频率也由每个月一种增加到了每个月 2 种甚至 4 种，有时候甚至有 10 种。

表 8-1 展示了主要编程语言的历史年表。

表 8-1 编程语言开发历史年表

1951	汇编语言
1954	FORTRAN (Formula Translator, 公式翻译器) 语言
1958	Lisp 语言 (List Processing, 表处理语言)
1959	COBOL 语言 (Common Business-Oriented Languages, 面向商业的通用语言)
1959	JOVIAL 语言 (Jules Own Version of the International Algorithmic Language, 国际算法语言的 Jules 版本)
1959	RPG 语言 (Report Program Generator, 报表程序生成器)
1960	ALGOL 语言 (Algorithmic Language, 算法语言)
1962	APL 语言 (A Programming Language, 一种编程语言)
1962	SIMULA 语言
1964	Basic 语言 (Beginner's all-purpose symbolic instruction code, 初学者通用指令代码)
1964	PL/I 语言
1964	CORAL 语言
1967	MUMPS 语言
1970	PASCAL 语言
1970	Prolog 语言
1970	Forth 语言
1972	C 语言
1978	SQL 语言 (Structured query language, 结构化查询语言)
1980	CHILL 语言
1980	dBASE II 语言

(续)

1982	SMALLTALK 语言
1983	Ada83 语言
1985	Quick Basic 语言
1985	Objective C 语言
1986	C++ 语言
1986	Eiffel 语言
1986	JavaScript 语言
1987	Visual Basic 语言
1987	PERL 语言
1989	HTML 语言 (Hypertext Markup Language, 超文本标记语言)
1993	AppleScript 语言
1995	Java 语言
1995	Ruby 语言
1999	XML 语言 (Extensible Markup Language, 可扩展标记语言)
2000	C# 语言
2000	ASP (Active Server Pages, 动态服务器页面) 语言
2002	ASP.NET 语言

表 8-1 展示的仅仅是所有编程语言中很小的一部分。我们将它展示在这里只是帮助那些不从事编程工作的读者了解一下目前已存在的编程语言是多么的丰富多彩。

如果你很熟悉编程工作,你应该可以从表 8-1 的列表中看到,编程语言的设计有两个不同的方向:

□ 对某些特定领域的问题处理做过优化的专业化语言,比如 FORTRAN、Lisp、ASP 及 SQL 语言。

□ 适用于多数领域的通用性语言,比如 Ada、Objective C、PL/I 及 Ruby 语言。

社会学研究发现,大多数专业化语言是由一个人或者两个人开发出来的,比如, Basic 语言是由 John Kemeny 和 Thomas Kurtz 开发的; C 语言由 Dennis Ritchie 开发; FORTRAN 由 John Backus 开发; Java 由 James Gosling 开发; 而 Brad Cox 和 Tom Love 开发出了 Objective C。

而那些通用性语言则通常是由专门的组织开发出来的,比如, COBOL 是由一个著名的委员会开发的,其主要工作是由来自美国海军的 Grace Hopper 来完成的;其他由委员会开发出来的语言包括 Ada 和 PL/I 语言。然而,也有一些通用性语言是由个人或者软件业同行开发出来的,比如 Ruby 和 Objective C。

程序员们似乎更愿意去尝试开发一些专业化语言,而不是尝试去开发一些如 PL/I 和 Ada 的通用性语言,这种现象的背后也许更多的是社会原因而不是技术原因。

这个话题值得从社会原因和技术原因两方面进行调查,因为 PL/I 和 Ada 语言看起来设计更加合理,结构更加健全,并且能够增强多种软件应用的功能,也获得了不错的反响。

在 20 世纪 70 年代末,出现了另外一种主流的对编程语言进行分类的方法,实际上,

与此相关的研究在这种分类出现之前就已经开始了。这种分类方式将语言分成两类，一类是面向对象语言，如 SMALLTALK、C++ 和 Objective C，另一类是没有使用面向对象方式及其相关术语的语言，如 Basic、Visual Basic 及 XML 语言。

在 2009 年的今天，在所有仍在使用的编程语言中，有超过 50% 属于面向对象语言的阵营，其他的编程语言多是过程化语言、函数式语言或者使用了其他操作方法的语言。

还有一种将编程语言分类的方法，其主要依据是编程语言是类型语言还是非类型语言。“类型语言”这个术语是指某种语言的操作仅对特定类型的数据有效。比如，一个类型语言不能对字符类数据执行数学运算操作。这类语言的例子有 Ruby、SMALLTALK 及 Lisp 语言。

相反的例子，即“非类型语言”是指这种语言的操作可以对任何类型的数据执行，这种语言的例子包括汇编语言和 Forth 语言。

类型语言和非类型语言的分类在某种程度上是不太明确的，类似的分类还有强类型语言和弱类型语言。除了分类上的模糊性之外，人们对于类型语言和非类型语言的优点和局限性还有一些争议。

8.4 编程语言普及性的探索

有许多种方法来研究各种编程语言的使用率和普及情况，包括：

1. 对于某种编程语言的网络搜索情况的统计学分析
2. 关于某种编程语言的出版书籍和文章的统计学分析
3. 各文献中对于某种编程语言的引用情况的统计学分析
4. 编程人员的招聘广告中所要求编程语言的统计学分析
5. 遗留应用所使用编程语言的调查和统计学分析
6. 新软件应用所使用编程语言的调查和统计学分析

一家名叫 Tiobe 的公司公布了一个编程语言普及性排行榜，其中包含了 100 种编程语言并且每月进行更新。本章写于 2009 年 5 月，表 8-2 列出了当月的编程语言排行榜中最受欢迎的 20 种语言。

表 8-2 2009 年 5 月的编程语言普及性排行榜

1	Java	8	JavaScript	15	RPG(OS/400)
2	C	9	Perl	16	ABAP
3	C++	10	Ruby	17	D
4	PHP	11	Delphi	18	MATLAB
5	Visual Basic	12	PL/SQL	19	Logo
6	Python	13	SAS	20	Lua
7	C#	14	PASCAL		

年长一些的读者可能会关心 COBOL、FORTRAN、PL/I 以及 Ada 语言在排行榜中的位置。在编程语言排行榜中，它们的排名比表 8-2 列出的靠后一些，大约在 21 到 40 名的位置。

由于每个月都会出现一种以上新的编程语言,各种语言的受欢迎程度在一个月之内也会有很大变化。每当一种有趣的编程语言出现时,它的受欢迎程度会大幅上升,但是经过长时间的检验,人们会考虑到这种语言是否实用,则它的受欢迎程度可能会下降得和当初上升得一样迅速。

编程语言的受欢迎程度和黄金时间段电视剧的流行程度有一定的相似性。新的电视剧,比如《好汉两个半》开始登上电视荧幕并吸引了数百万观众,这种流行性可能会持续好几季。少数电视剧,比如《宋飞正传》,广受观众欢迎因而多家媒体争相购买,甚至在结束拍摄之后很多年仍在播出。但是很多电视剧在拍摄一季之后便销声匿迹了^①。

有趣的是,对于编程语言生命周期的预期和对于电视剧生命周期的预期几乎是一样的。很多编程语言在活跃了大约数“季”之后便消失在人们的视野中,其他的语言则成为了行业标准而持续活跃了许多年。然而,如果我们对所有2500种编程语言整体考虑的话,每一门编程语言用于新软件开发的平均活跃生命周期是不到5年的时间,极少数的语言对于编程人员的吸引力能够保持10年以上。

有些编程语言可以跻身像美剧《宋飞正传》和《我爱露西》一样的级别^②,这些语言大概已经保持了超过25年之久的行业巨头的位置,它们包括:

- ☐ Ada
- ☐ C
- ☐ C++
- ☐ COBOL
- ☐ Java
- ☐ Objective C
- ☐ PL/I
- ☐ SQL
- ☐ Visual Basic
- ☐ XML

在讨论编程语言时,“行业巨头”这个名词是指这样的一些语言,它们已经不再由其创制者直接控制其发展,而是由某个用户组或者某个商业公司来进行控制,或者这些语言已经被置于公开环境下因而可以使用一些开源的编译器。

如果我们能够对使用这些长期生存的编程语言进行开发的软件应用的数量进行统计分析并形成参照基准的话,那将会是非常有意思和有价值的资料。毫无疑问,在全球范围内,使用C语言或COBOL语言进行开发的软件应用数量都超过了100万个。

① 由于本书的原作者是美国人,他讲述的电视剧也是美国商业电视剧。美国商业电视以每年9月中旬至第二年4月下旬为一个播出季,新季以美国电视艺术与科学协会主办的“艾美奖”颁奖典礼为序幕。美国电视剧一般一个星期只播一集,并且是边拍边播的,他们很注重收视率,一部收视率低下的电视剧是无法生存的,只要吸引不了观众的注意力,那么不管该剧的情节进行到何处,电视台都毫不留情地停拍。原文中提到的《好汉两个半》、《宋飞正传》都是美剧的名字。——译者注

② 这两个都是美剧中经久不衰的经典之作。——译者注

事实上,如果我们继续进行前面和娱乐行业的类比,那么我们可以设置一个奖项颁发给那些应用到大量软件开发中的语言,这将会是一个非常有意思的事情。也许,我们的银奖要颁发给那些应用超过 10 万个软件的开发语言,金奖颁发给应用超过 100 万个的语言,而“白金唱片”的荣誉属于那些超过 1000 万个软件使用的开发语言。

如果我们真的要建立这样一个奖项,我们可以给它一个很好的名字叫做 Hopper 奖,这个名字以 Grace Hopper 少将命名,她对于编程语言的发展做出了极大的贡献,尤其是对于 COBOL 语言。事实上,COBOL 可能是历史上第一个进入“百万应用俱乐部”的编程语言。

尽管为各种软件所应用的语言颁奖这个想法听起来很有趣,但是这意味着我们需要弄清对于某种特定语言来说,有多少软件是使用它来开发的,或者使用了包括这个语言在内的多种语言,并得到所有的统计数据。截至 2009 年,软件行业并没有这样的数据。

令人惊奇的是,对于某种特定的软件应用来说,选择使用哪种语言进行开发是一件完全主观的事情。在某次会议上,有人询问一位 IBM 的同行是否曾经使用过 APL 语言进行开发,他的回答是:“不,我不喜欢这个语言。”

我们需要设计一种方法来描述和记录各种编程语言的功能特性并将之编辑成目录,这在技术上是可行的。超过 2500 种编程语言的存在,使得我们确实急需这样的一个目录。即使这份目录在开始阶段只能记录 100 种最广泛使用的语言,仍然能够提供非常有价值的信息。

建立一个有效的编程语言分类系统所需要考虑的全部方面并不在本书的讨论范围之内,但这样的分类系统可能需要包含如下的一些因素:

编号	分类系统中的因素	对此因素的解释或示例
1	语言名称	编程语言的名字
2	架构	面向对象语言、功能性语言、流程化语言等
3	起源	创立年代、发明者姓名
4	资料来源	编译器的发布链接(URL)
5	当前版本	当前版本号,如第一版、第二版或者其他版本
6	技术支持	负责维护的组织的 URL 或地址
7	使用者协会	使用者群组的名称、URL 以及地址
8	指导材料	该语言的书籍或相关学习资源
9	评论	在相关期刊上发表的对该语言的评论
10	法律方面的状态	开源、使用许可证、专利等
11	语言定义	语言是否是正式的编程语言
12	语法规则	语法规则的描述
13	语言类型	强类型、弱类型、非类型等
14	专注领域	数学、网络、嵌入式、图形处理等
15	硬件平台	该语言支持的硬件平台
16	操作系统平台	该语言编译器运行的操作系统
17	使用倾向	所应用的目标软件的类型
18	已知的使用局限性	性能、安全性、特定领域的问题等
19	变体语言	在该语言基础上的变种语言

(续)

编号	分类系统中的因素	对此因素的解释或示例
20	同伴语言	.NET 语言、XML 语言等 (和该语言一起联合使用的语言)
21	扩展	由该语言的使用者添加的指令
22	等级	相对于汇编语言的逻辑代码行数量
23	逆火等级	每个功能点的逻辑代码行数量
24	可重用代码来源	认证模块、未认证等
25	安全特性	固有的安全特性, 如 E 语言内的安全特性
26	调试器是否可用	调试器名称
27	静态分析是否可以应用	静态分析工具名称
28	开发工具是否可用	开发工具名称
29	维护工具是否可用	维护工具名称
30	当前使用该语言开发的软件数量	大约 100、1000、10 000 或者 100 000 等

存在如此大量的编程语言却没有一个标准的分类系统真的是一件让人非常惊讶的事情。当我们使用诸如“编程语言分类系统”(Taxonomies of programming languages)或“编程语言类别”(Categories of programming languages)的关键词在网络上进行搜索时,我们可以找到数十个讨论是关于需要考虑的方面。然而,这些讨论的结论千差万别,有些甚至包含 50 多种分类方式,但看起来缺少一些基础的有组织性的原则。

回到我们最主要的话题上来,在某种程度上,编程语言有一种令人担心的现象,那就是很多软件应用的预期使用寿命比它们所使用的编程语言的预期寿命还要长。其中的一个例子就是美国退伍军人管理局负责维护的医疗病历系统。这个系统是使用 MUMPS 语言^①开发的,而且这个系统的生存时间比 MUMPS 语言还要长。

对于软件工程经济学的学习者来说,很显然的一个事实就是,如果一门编程语言的平均预期寿命是 5 年,而一个大型软件的平均预期使用寿命是 25 年的话,那么软件的维护费用会变得比预期要高,因为相当多的历史软件都是使用已经死掉或者即将死掉的编程语言编写的。

8.5 我们到底需要多少种编程语言

编程语言的过剩引发了一个需要在软件工程文献中讨论的基本问题,那就是我们到底需要多少种编程语言?

我们已经有了数千种编程语言,这种情况必然会引起人们的一个疑问:世界上存在超过 2500 种编程语言,这到底是好事还是坏事?

有一种观点坚持认为数千种编程语言的存在是一件好事,其观点围绕着这样的一个事

① MUMPS 语言,简称 M 技术,全称 Massachusetts General Hospital Utility Multi-Programming System,麻省总医院多用途程序设计系统,与 FORTRAN 和 COBOL 属于同时代的语言。因为这门语言最主要是用于医疗数据库方面,所以其应用并不像 SQL Server、Oracle 等那么广泛,在国外的医疗行业应用较多。——译者注

实,那就是编程语言是为解决某些特定种类的问题做过优化的。由于我们会不断遇到新问题,我们也会不断地需要新语言的出现,至少我们应该假设我们会需要新的语言。

而另一种观点坚持认为存在数千种语言并不是一件好事,其主要的理由围绕经济学原因展开。维护那些使用已经死掉的语言所开发的遗留应用所需费用必然十分高昂,这会成为每个公司的噩梦。同时,随着新语言的出现,公司必然要持续地投入以培训编程开发人员学习最新最热门的语言,这也将产生不菲的开销。在收集经过认证的可重用代码时,如果我们需要考虑数千种语言,那么收集如此大量的信息不仅更加困难,而且费用也必将更加昂贵。

数千种编程语言的存在导致了软件工程的一个新分支行业的出现。这个分支行业主要考虑的是如何把死掉的或即将死掉的编程语言转换为新的比较活跃的语言。比如,将1967年左右的MUMPS语言转化为C语言或者Java语言已经成为可能,并且可能使转换过程自动化完成。

此外,还有一个必然出现的分支行业,那就是对遗留应用进行“改造”或者定期进行专门维护活动。这些维护的主要目的包括清除已经无用的代码、移除容易产生错误的模块等,同时,由于我们会不断地对这些应用进行小的更改,随着时间的过去,软件应用的基本复杂度和循环复杂度也必然会增加,维护工作也需要尽量去减少软件的复杂度。

语言学家和一些熟悉人类自然语言的人们已经注意到,一种语言翻译成为另外一种语言时,其翻译结果并不是完美的。比如,在一些爱斯基摩人的方言中,有超过30个不同的词语来描述多种多样的雪。如果我们要将它翻译成其他语言,比如英语,由于这门语言源于温和气候的地区,其语言词汇中对于雪的变化描述也很少,我们就很难找到爱斯基摩方言中这些词汇的准确翻译。

由于许多编程语言为某些特定种类的问题定制了专门的语言结构,如果我们将一种语言翻译到其他语言,由于原语言和翻译后的语言结构不同,就可能会出现一些非常难以处理的结构,在维护及修订功能的时候,这种结构对于编程人员来讲可能既难以理解,更无法进行处理。但尽管如此,如果这种翻译能够打开这门死掉的语言通向大量静态分析工具和维护工具的大门,那么花在翻译上的工作很可能是值得的。

要讨论我们到底需要多少种编程语言这个问题,我们有必要考虑这样一个方面,那就是我们需要使用计算机所解决的问题所属的领域。我们可以将这些问题总结为10个不相关的领域并分为不同的两个处理类型,如表8-3所示。

表中所展示的两大领域反映了现今存在软件的主要形式:(1)处理信息的软件;(2)控制物理设备的软件或者处理物理属性(如声音、光或音乐等)的软件。

这两个非常宽泛的分类可能会导致这样一个结论,那就是也许我们最少需要两种编程语言来处理所有领域的问题:一种语言针对信息处理系统进行优化;另一种对处理物理设备或电信号进行优化。然

表 8-3 软件应用所处理的问题领域

逻辑及数学领域	物理装置领域
1. 数学计算	1. 基于传感器的电信号
2. 逻辑及运算法则表达	2. 声音信号及音乐
3. 数值型数据	3. 静态图像
4. 文本及字符串数据	4. 动态图像
5. 时间及日期数据	5. 颜色

而,对于一些通用性编程语言(如 PL/I 和 Ada 语言)的跟踪记录表明,这些语言在试图同时处理多种类型的问题上并不那么成功。

只有极少数问题是非常“单纯”地只处理某一领域的事情,事实上,大多数的软件应用处理的问题都是各个领域的混合体。这不禁让我们得出了这样一个结论,那就是很可能编程语言所反映的问题是多个领域的一种排列组合,而不是某个单独的领域。

如果我们考虑所有 10 个领域的排列组合,那我们最终的结论可能是 3 628 800 种编程语言。这甚至比建立一个可以处理所有领域问题的超级语言更加不太可能发生。

通过对一些样例的分析(这些样例既包括信息处理软件,也包括嵌入式软件和系统软件),我们在上述 10 个领域中挑选出了 4 个。我们可以暂时假设一个典型软件应用需要处理这 4 类不同的问题。通过对这 4 个领域进行排列组合,我们得到了这样一个假设,那就是软件工程领域最终需要 5040 种不同的编程语言。

截至 2009 年,我们已经拥有了大约 2500 种不同的编程语言及其派生语言,因此,将来我们大约还需要开发出另外 2500 种语言。考虑到新语言出现的频率大约是每年 100 种,我们可以预期,新的编程语言会以这样的速率出现并维持大约 25 年的时间。从经济学的立场来看,这似乎并不是一个能够有效节省开支的解决方案。

假设软件工程社区最终拥有了 5040 种编程语言,那么这些语言的分布可能是这样的:

□ 4 800 种语言已经死掉,或者即将死掉,只有极少数程序员在使用。

□ 200 种语言在遗留应用中被使用,因此需要进行维护。

□ 40 种是新的语言,并且正在吸引越来越多的程序员使用。

如果我们想为各种即将出现的新问题快速制造出 2500 种专门语言的话,有一个技术上的备选方案,那就是考虑制造一个复合编译器,它可以支持对各种领域的问题进行排列组合。

8.6 建立一个国家级的编程语言翻译中心

当我们考虑快速制造出另外 2500 种编程语言的备选方案时,建立一个正式的语言翻译中心也许是有价值的。这个翻译中心应该收藏所有已知编程语言的定义。

这个翻译中心可以为语言翻译提供指导,以便人们可以将已经死掉的或即将死掉的语言翻译成为现在比较热门的语言。一些公司已经开始实施这样的翻译了,但是在今天所有的 2500 种语言中,只有很少数语言的翻译在技术和语言学上都是非常精确的。截至 2009 年,自动化的翻译可能只能处理 2500 种语言中的大约 50 种。

考虑到现在已经存在的语言数量之大以及新的语言诞生速度之快,这样的—个翻译中心大约需要 50 名全职工作的员工。这就意味着只有大型公司,如 IBM、微软,或者大型政府机构如国土安全部、国防部,才有能力去做这样的尝试。

除了进行翻译工作,国家编程语言翻译中心还可以对所有已知语言进行仔细的语法分析,以便能够确定各个语言的主要长处和弱点。大多数编程语言的一个明显弱点就是都不安全。

翻译中心的另一个功能就是记录各个语言的统计信息,包括使用该语言的软件应用数

量以及哪些种类的软件在使用。比如，财务系统软件、武器系统软件、医疗系统软件、税收系统软件以及病历记录系统软件所使用的语言，这些信息具有重要的商业价值和政治价值。

表 8-4 总结了 40 种对于美国来讲具有绝对重要性的软件，同时，表 8-4 也向我们展示了这 40 种软件所使用的各种编程语言。代码翻译中心的一个主要职责，就是积累更清晰准确的关于重要软件类型及其所使用语言的数据。

表 8-4 的两列数据都需要进行进一步的调查研究。毫无疑问，重要的软件应用类型显然比这多。同时，为了使表格能够在一页纸上面打印出来，表格的第二列（即语言列表）仅仅列出了 6~7 种语言。就美国范围来说，许多重要软件使用了 50 种甚至更多的语言。

据笔者所知，商务部的北美工业分类（North American Industry Classification）代码列出了会制造大量软件的至少 250 类行业。然而，表 8-4 中列出的 40 种主要行业大约包含了那些对美国商业和政府运作具有重要作用的软件中的一半。

表 8-4 重要软件所使用的开发语言列表

重要软件类型	开发语言
1. 航空管理	Ada、汇编语言、C、Jovial、PL/I
2. 反病毒软件 & 安全软件	ActiveX、C、C++、Oberon7
3. 汽车工业	C、C++、Forth、Giotto
4. 银行业	C、COBOL、E、HTML、Java、PL/I、SQL、XML
5. 宽带网络服务业	C、C++、CESOF、Java
6. 手机业	C、C++、C#、Objective C
7. 信用卡	ASP.NET、C、COBOL、Java、Perl、PHP、PL/I
8. 信用查询	ABAP、COBOL、FORTRAN、PL/I、XML
9. 信用合作社	C、COBOL、HTML、PL/I、SQL
10. 犯罪记录	ABAP、C、COBOL、FORTRAN、Hancock
11. 国防工业	Ada、汇编语言、C、CMS2、FORTRAN、Java、Jovial、SPL
12. 电力能源	汇编语言、C、DCOPEJ、Java、Matpower
13. FBI、CIA、NSA 等	Ada、APL、汇编语言、C、C++、FORTRAN、Hancock
14. 联邦税收	C、COBOL、Delphi、FORTRAN、Java、SQL
15. 航班管理	Ada、汇编语言、C、C++、C#、LabView
16. 保险业	ABAP、COBOL、FORTRAN、Java、PL/I
17. 邮政及快递	COBOL、dBase2、PL/I、Python、SQL
18. 制造业	AML、APT、C、Forth、Lua、RLL
19. 医疗器械	汇编语言、Basic、C、CO、CMS2、Java
20. 医疗记录	ABAP、COBOL、MUMPS、SQL
21. 医疗保险	汇编语言、COBOL、Java、PL/I、dBase2、SQL
22. 市政税收	C、COBOL、Delphi、Java
23. 航海业	汇编语言、C、C++、C#、Lua、Logo、MatLab
24. 石油能源	AMPL、C、G、GAMS/MPGGE、SLP
25. 开源软件	C、C++、JavaScript、Python、Suneido、XUL
26. 大型操作系统	汇编语言、C、C#、Objective C、PL/S、VB

(续)

重要软件类型	开发语言
27. 小型操作系统	C, C++, Objective C, OS/2, SR
28. 制药	C, C++, Java, PASCAL, SAS, Visual Basic
29. 治安记录	C, COBOL, dBase2, Hancock, SQL
30. 卫星工业	C, C++, C#, Java, Jovial, PHP, Pluto
31. 有价证券交易	ABAP, C #, COBOL, dBase2, Java, SQL
32. 社会安全	汇编语言, COBOL, PL/I, dBase2, SQL
33. 州税收	C, COBOL, Delphi, FORTRAN, Java, SQL
34. 陆地运输	C, C++, COBOL, FORTRAN, HTML, SQL
35. 有线电话	C, CHILL, CORAL, Erlang, ESPL1, ESTEREL
36. 电视广播	C, C++, C#, Java, Forth
37. 选举设备	Ada, C, C++, Java
38. 武器系统	Ada, Assembly, C, C++, Jovial
39. 网络应用	AppleScript, ASP, CMM, Dylan, E, Perl, PHP, .NET
40. 福利系统(州政府)	ASP.NET, C, COBOL, dBASE2, PL/I, SQL

这40种软件类型对于美国商业和政府具有非常重要的作用。也正因为这种重要性,在各种形式的网络攻击中,有大约75%的攻击是针对以上类型的软件,这些攻击的形式包括病毒入侵、间谍软件、搜索爬虫以及拒绝服务攻击(DoS attack)。这40个行业需要将软件开发的重点放在安全性上面。即使仅对以上40个行业所使用的编程语言做一个粗略的检查,我们几乎没有发现任何一门语言对于病毒或者是恶意软件有特别的抵抗力。

对于所有的40种软件,维护费用都是昂贵的。而对于其中许多软件来讲,还将会变得越来越昂贵。这主要是因为我们需要持续地进行维护,而这些软件使用了许多种不同语言进行开发,这种维护将会非常困难。

将一种相对古老的编程语言翻译为另一种新语言还有一个技术上的副产品,那就是在翻译旧语言的同时消除其中安全性方面的弱点,这也会为国家编程语言翻译中心的工作增加价值。

如果语言翻译中心像一个营利的商业组织那样运作,它很可能会发展成为一个具有相当规模的公司。如果我们假设这家公司和当初解决千年虫问题的公司使用相同收费标准的话(大约是每行代码1美元),在假设拥有足够准确的翻译技术的前提下,国家翻译中心每年大约会有7500万美元的营业额。

笔者的建议是,我们不能继续放任新编程语言在随机的时间段内快速地开发出来,而应该从基础语言学的角度去定义编程语言。

一个由语言学家、软件工程师以及各行业专家组成的研究小组也许可以用最有效的方式来定义如何去描述我们之前提到的10个问题领域,并定义这些领域的排列组合。这个小组的目标,就是要研究出我们所需要的编程语言的最小集合,而这个集合可以解决这10个领域任何组合所导致的问题。

如果经济学家也加入到这个小组中，他们可以对那些使用了数百种已经死掉的或者即将死掉的语言所开发的软件进行研究，以便研究出试图维护及偶尔更新这些软件对于公司财务会有多大的影响。

8.7 为什么大多数软件都使用 2 ~ 15 种编程语言

软件工程中有一个异乎寻常的现象，那就是同一个软件应用中出现了多种编程语言。这并不是一个新的潮流，很多以前的软件也会使用几种语言的组合，如 COBOL 和 SQL 语言。而近期的类似组合中可能会包括 Java 和 HTML 或者 XML。

与之类似的现象是，很多编程语言自身就是其他两种或多种语言的组合体。比如，Objective C 语言就混合了 SMALLTALK 和 C 语言的功能特性，而 Ruby 语言则混合了多种语言的功能特性，包括 Ada、Eiffel、Perl、Python 等。

回想一下，大多数的编程语言都是在一定程度上专注于某个领域的，而这些语言通常都比那些通用性语言要更流行一些。那么如果我们假设问题所涉及的领域比某种单独的编程语言所能够解决的领域要多的话，我们就可以解释为什么软件会使用多种不同的语言了。

前面曾经提到过，许多软件应用通常会涉及表 8-3 中所列 10 个领域中的 4 个。然而，许多编程语言看起来只针对其中的 1 到 3 个领域进行了优化。这就造成了这样一种状况，为了解决软件所涉及的所有问题，我们需要使用多种编程语言来满足这种需要。

当然，如果我们使用任何一种通用性语言，如 Ada 或者 PL/I 语言，则可以帮助我们减少所使用的语言数目，但是由于一些社会学原因，这些通用性语言并不像那些专门性语言那样受人欢迎。

同一个软件应用中包含了多种不同语言这一现象还意味着，那就是其开发会更困难、调试会更困难、静态分析会更困难、代码审查也会更加困难。而产品投入使用之后，对产品的维护和升级工作也会变得更加困难。

表 8-5 向我们举例说明了随着单个软件应用所使用语言的增加，开发和维护费用的增长情况。这些费用是通过和仅使用一种语言的情况进行对比所得到的。^①

表 8-5 使用多种语言对于软件费用的影响

软件中包含的语言数目	开发费用	维护费用	软件中包含的语言数目	开发费用	维护费用
1	1.00 美元	1.00 美元	6	1.22 美元	1.30 美元
2	1.07 美元	1.14 美元	7	1.23 美元	1.35 美元
3	1.12 美元	1.17 美元	8	1.27 美元	1.40 美元
4	1.13 美元	1.20 美元	9	1.30 美元	1.47 美元
5	1.18 美元	1.24 美元	10	1.34 美元	1.55 美元

随着软件中所使用语言数目的增加，开发费用和维护费用都会增加，但是维护费用受到的影响更加严重。

① 即我们假设同一个软件，仅使用一种语言进行开发则其费用为 1 美元，从而得出使用多种语言进行开发时的费用。——译者注

8.8 有多少程序员使用多种编程语言

对于软件中所使用的语言或者程序员的数量并没有一个十分真实确切的统计，尽管美国商务部和劳工统计局确实曾经就以上话题发布过报告，但众所周知那份报告并不准确。

几年前，笔者和同事们进行的一项调查显示，大多数大型企业的人力资源部门并不清楚公司内部到底有多少编程人员或是软件工程师。由于政府的数据是根据人力资源部门提供的报告进行统计的，如果人力资源部门自身也不清楚，那么他们也不可能向政府部门提供准确的数据。

政府部门统计报告可能过于保守地估计了程序员和软件工程师的数量，其原因就是公司使用了含糊的职位名称。比如，一些大型公司使用“技术人员”这样一个名称作为职位总称，其范围可能包括了软件工程师、硬件工程师、系统分析人员以及可能其他数十种职位。

想要了解有多少软件工程师还有另外一个问题，那就是许多在嵌入式软件项目中工作的员工实际上并不是经过培训的软件工程师或者计算机科学家，而是电力工程师、航天工程师、电信工程师或者其他种类的工程师。

由于这些更有历史的工程师比软件工程师的地位要高，许多在嵌入式软件项目中工作的人们拒绝称自己为软件工程师，而更愿意坚持自己真正的学术领域的称呼。

笔者和同事曾经进行过一项对大型软件密集型公司（如 IBM、AT&T、Hartford Insurance 等）的调查，得到了这些公司所雇用的软件专家（如质量保证人员、数据库管理员等）数量的信息。

这项调查的形式包括进入所调查公司内部同人力资源部门、当地软件工程师经理以及企业高管们进行讨论。在和当地经理及高管们进行讨论时我们发现，没有任何一个人力资源部门准确地统计过软件工程师的数量。

基于对以上目标公司的现场访谈数据，并使用其数据来推断全国的情况，笔者推断在 2009 年左右，美国国内软件工程师的总数量大约是 250 万。而截至 2009 年，政府的统计数据 displays 美国国内大约有 60 万程序员，但是由于上面已经说过的原因，这个数据比实际的要低。另外，政府统计数据会倾向于忽略掉那些只有一个人的公司以及那些开发小型软件或独立软件的个人。

在这些软件工程师中，大约有 60% 的人从事维护软件和增强现有软件功能的工作，而其余 40% 的人从事新软件开发工作。当然，不同的行业之间会有区别。比如，对于网络应用来说，开发人员比维护人员更多，因为这些软件都是相当年轻；但是对于传统大型机上的商业软件、常见的嵌入式软件和系统软件来说，维护人员数量会比开发人员数量多出一个很大的数值。

表 8-6 展示了在美国使用不同语言的软件工程师的数量。然而，表 8-6 中的数据只是一个大致的估计，并不十分准确。数据不准确的原因是，许多软件工程师了解多种编程语言而且在工作中使用多种语言。

表 8-6 使用不同语言的软件工程师数量的粗略估计

开发语言	软件工程师数量	维护语言	软件工程师数量
Java	175 000	COBOL	575 000
C	150 000	PL/I	125 000
C++	130 000	Ada	100 000
Visual Basic	100 000	Visual Basic	75 000
C #	90 000	RPG	75 000
Ruby	65 000	Basic	75 000
JavaScript	50 000	汇编语言	75 000
Perl	30 000	C	75 000
Python	20 000	FORTRAN	65 000
COBOL	15 000	Java	60 000
PHP	15 000	JavaScript	40 000
Objective C	10 000	Jovial	10 000
其他	15 0000	其他	150 000
总计	10 00000	总计	1 500 000

尽管如此，表 8-6 的数据仍然说明了一个关键点：软件开发中最常使用的语言 and 软件维护最常使用的语言并不相同。这种情况为软件工业带来了许多麻烦。

表 8-6 可以说明的最明显问题就是，让负责开发的员工去从事维护工作是一件很困难的事情，因为人们会有一种观点，认为旧的语言并不如新的现代编程语言那么有魅力。

第二个问题就是，由于新软件开发和已有软件维护工作所使用的语言不同，我们很可能需要使用两套不同的工具来工作。从事开发工作的员工会对于新的现代工具更感兴趣，这些工具包括静态分析工具、自动化测试工具以及其他各种有足够创新的产品。

然而，这些新工具中的大部分并不支持旧的编程语言，因此软件维护团队需要使用包含各种不同工具的维护工作台来进行工作。比如，软件维护人员会较多使用分析软件的循环复杂度和基本复杂度的工具，而新产品的开发工作并不经常使用这些工具。对软件执行流程进行追踪的工具也会更广泛地应用于软件维护工作而不是开发工作。另外一种对于维护工作有帮助的工具可以对遗留应用的代码进行挖掘从而提取出那些隐藏的业务规则。还有一种新工具可以帮助维护工作，它可以通过分析代码的语法从而自动计算出软件的功能点数量。

对于程序员来讲，学习一门新编程语言是一件相当简单的事情，但是恐怕没有人可以掌握 2500 种语言。通常来讲，美国国内的一名程序员可能会对某一门语言相当精通，同时比较熟悉另外的三门语言。有些人也许可以掌握多达 10 门语言。很明显，编程语言种类的过剩为进行理论培训制造了很大的困难，同时也为程序员保持其技能的前沿性制造了障碍。

概括来说，开发人员所用工具和维护人员所用工具有很大的不同，这在很大程度上是因为开发人员和维护人员所使用编程语言的不同。

由于在 2009 年的今天，人们广泛使用的编程语言将会在 10 年内退出历史舞台，软件

维护工作将会面对非常严重的挑战。

新开发语言以高于每个月两种的速度不断出现。在这些语言当中，大多数的寿命会非常短暂。然而，大多数使用这些短命的语言所开发的软件会被使用许多年。因此，这些和遗留应用有关的编程语言都需要进行维护，这些语言的集合会不断增长，其增长速度有时甚至会达到每年 50 种新语言。

数千种编程语言的存在会导致一个经济学的问题，那就是编程语言的过多导致维护费用在不断提高。具有讽刺意味的是，新编程语言的一项声明就是“这门语言可以提高开发生产率”。即使我们假定这样的声明是正确的，它也仅在开发过程中是正确的。每一种新的语言最终都会成为软件维护的负担。这是因为软件自身往往比它们所使用的开发语言有更长久的寿命。今天的“新”语言会一个个地退出，而仅仅留下数百个过时的遗留应用，而伴随着这些遗留应用的，只有日渐稀少的程序员、效率低下的工具和有时甚至无法工作的编译器。

8.9 源代码中通常会出现何种类型的缺陷

在 2008 年和 2009 年，有人就软件缺陷的类型进行了一次专门调查并总结了常见的 25 种严重缺陷。这项调查是由美国系统网络安全协会（SysAdmin Audit Network Security, SANS）发起，并由包括 MITRE 在内的约 30 家组织机构合作完成的。

这项新调查自然引起了很多人的关注。毫无疑问，这份调查报告已经成为软件质量和软件安全性历史上一份具有里程碑意义的报告。实事求是地讲，所有的软件工程相关组织都应该保存这份报告并要求公司内部相关员工研读，不仅包括软件工程师、质量保证人员，还包括软件部门经理和高级管理人员，都应研读。

人们可以通过 SANS 的网站或者 MITRE 公司的网站获取这份报告。相关的 URL 是：

- www.SANS.org
- www.CWE-MITRE.org

尽管软件工程师如今已经成为正式的职业并已经开发出数百万个软件应用，但直到最近人们才开始正视软件源代码中的缺陷，并开始专注于研究这些缺陷的特点。在这方面，SANS 的报告具有非常重要的意义，因为这份报告是由来自各大软件组织的 40 名专家起草完成的，因此这份报告中所述的问题具有普遍意义，而不是仅仅针对某一家公司。

多年以来，如 IBM、AT&T、微软或 Unisys 这样的大型公司都有相当成熟的缺陷跟踪监控系统，同时，这些公司也会使用根本原因分析方法来分析所发现的缺陷。他们曾经对外公开了一部分这些内部缺陷跟踪系统的统计结果，但通常情况下人们并不认为这些数据具有广泛的适用性。

人们很早就知道了一些软件中的常见错误，如缓存区溢出、错误的分支语句或者遗漏了容错处理程序。这些错误是众所周知的，因而有经验的软件工程师可以避免它们出现。但如果我们想要对代码缺陷进行缜密的分析并进行量化，这些显然是不够的。

在对缺陷进行分析和量化方面，SANS 协会的报告为我们提供了一个非常好的例子，来

自多家公司的顶尖专家通过协同工作的方式探讨并发现了一些通用的问题。SANS 协会报告小组包括了来自学院、政府及商业公司的专家。同时，他们也积极促进了这三类组织的成功合作。通常情况下，这三类组织之间并不会合作，而是具有对抗性质的，因此联合这三者协同合作并完成一份非常有价值的报告是一项难得的成就。

我们希望这份报告的工作模式会成为未来协同工作的模式，使用这种方式，我们可以对软件工程中其他的重要问题进行研究。有一些方面的问题使用这种合作模式可能会得到很好的结果，这些问题包括：

1. 缺陷去除方法
2. 软件开发的经济效益分析
3. 软件维护的经济效益分析
4. 软件规模度和测算
5. 软件的可重用性

部分 SANS 研究小组参与机构的名单如下（按字母顺序排列）：

- 苹果公司 (Apple)
- Aspect Security
- Breach Security
- CERT
- 美国国土安全部 (Homeland Security)
- 微软公司 (Microsoft)
- MITRE 公司
- 美国国家安全局 (National Security Agency)
- 甲骨文公司 (Oracle)
- 普渡大学 (Perdue University)
- Red Hat 公司
- 塔塔集团 (Tata)
- 加州大学 (University of California)

以上只是报告参与者的部分名单，但是由此我们可以看出，这份报告的研究小组包括了学术组织、商业软件组织以及政府机构。

这份报告将所提到的 25 类安全问题分成了三个主要领域。我们强烈推荐本书的读者阅读这份报告，所以在此我们仅列出这 25 类问题的名称：

交互领域

1. 输入数据缺乏验证
2. 输出信息未经加密
3. SQL 查询的结构
4. 网页页面结构
5. 操作系统命令的结构

6. 敏感数据的公开传输
7. 伪造不同网站间的访问请求
8. 竞争条件
9. 错误信息中的漏洞

资源管理领域

10. 无限制使用内存缓存
11. 状态数据失去控制
12. 对路径和文件名缺乏控制
13. 危险路径
14. 未进行源代码版本控制
15. 使用未经验证的重用代码
16. 未经检查即释放资源
17. 初始化信息未加验证
18. 计算错误

防护漏洞

19. 缺乏用户验证和访问控制
20. 加密算法强度不足
21. 使用“硬编码”设置和储存密码
22. 危险的权限分配
23. 随机化算法强度不足
24. 系统特权的随意发布和滥用
25. 客户机/服务器模式中的安全失误

完整的 SANS 研究报告包含了以上 25 种缺陷的详细信息，同时还提供了一些附加信息，包括这些缺陷最常见形式、预防方法等重要内容。这也是我们极力推荐读者去阅读完整 SANS 报告的原因。

截至 2009 年，这 25 类问题可能在超过 85% 的运营软件中都存在。而在所有成功的恶意软件攻击中，人们在超过 95% 的情况中发现了以上 25 类问题中的一种或多种。无须多讲，SANS 报告是一份非常重要的文档，值得人们广为传播和深入研究。

对于所有引入了软件测试、静态分析、审查以及质量保证的公司来讲，SANS 报告是一份有价值的材料。它为我们提供了一份可靠的检查清单，其中包括了软件在发布之前需要检查的方面，以便我们的产品可以安全地投向外部市场。

8.10 软件缺陷的逻辑和属性

SANS 研究所的报告为我们准确定义了软件代码中的严重缺陷，尽管如此，在我们将软件产品交付到用户手中后，一旦缺陷再被发现出来，我们还需要讨论并定义一些附加的缺陷信息。以下列表讨论了一些和软件缺陷相关的逻辑问题和属性：

1. **缺陷** 人们在使用软件时出现的问题，可能是软件停止工作或出现了不正确的运行结果。缺陷可能源自错误的程序行为（即开发人员错误地编写了这部分功能）或者源自一些疏漏（即开发人员无意中漏掉了某个特殊情况下的处理动作）。

2. **缺陷严重等级**（源自 IBM 的定义）严重等级 1：软件停止工作；严重等级 2：重要功能不工作或行为不正确；严重等级 3：次要功能出现问题；严重等级 4：对软件使用没有影响的错误。

3. **无效缺陷** 软件工程师将一个问题归为软件缺陷，但后续的分析却发现这个问题是由软件自身之外的因素所导致的。最常见的无效缺陷类型包括硬件问题、用户操作错误或将操作系统所导致的错误误认为是软件错误。这些缺陷总计起来可占所有发现的有效缺陷的 15%。

4. **暂缓处理的缺陷**（IBM 术语）某个特定用户针对一款软件报告了一个缺陷，但是除了这个用户所使用的这一版本之外，在这款软件的其他版本上并不能重现这个问题。通常情况下，这类缺陷是由于在某些特定的硬件设备上同时运行了某些特定的软件组合所导致的。这种情况通常很少见，而且即使出现了也很难进行跟踪和修复。

5. **误报** 静态分析工具或者某个测试用例认定某段代码中包含某种隐含的缺陷。但进一步的分析证明这段代码其实是正确的。

6. **衍生缺陷** 软件中的某个并不是由开发团队的任何错误所导致的缺陷，其真实原因是该缺陷由开发团队所使用的编译器或开发工具所导致。这类缺陷的例子包括代码生成器中的错误或者自动化测试工具的错误。开发人员使用这些工具时非常信任它们，但结果是，这些工具产生了问题。衍生缺陷的一个例子就是编译器未能正确处理某条指令所导致的问题。编译器完成了对代码的编译和执行，但是指令并没有像编程语言说明书中所定义的方式运行。这时我们有必要去查看编译器所产生机器语言指令列表以确定这个衍生缺陷，因为通常这种缺陷是无法从源代码中发现的。

7. **漏报的缺陷** 这种缺陷和衍生缺陷有一定的相似之处，但其原因可能是测试覆盖范围不完整或者静态分析工具的遗漏。测试系列活动对于任何软件应用的代码覆盖率都不会达到 100%，对于有些大型软件系统来说甚至可能低于 60%，这一点是广为人知的。为了减少漏报缺陷或不完整测试的影响，我们有必要使用一些测试覆盖率分析工具。如果测试覆盖率与期望差距较大，我们可能需要进行一些特殊的测试或者引入正式审查。

8. **数据缺陷** 这类缺陷并不是由软件的源代码所导致，而是由软件的输入输出数据所导致。一个很常见的数据缺陷就是不正确的邮件地址所导致的软件问题。数据缺陷通常数量很庞大，而且有可能会很严重。同时，这类缺陷也很难修复。数据缺陷的数量很可能比代码缺陷还要多，但到底是哪方面的责任所导致这样的缺陷却很难说清。数据缺陷的一个更加严重的例子是信用报告中的错误。这种错误会降低用户的信用评级，但却不会给出任何合理的原因，同时软件中也不会出现明显的错误和缺陷。数据错误的修复难度已经是恶名远扬了，这种现象部分是因为没有任何有效的质量保证组织参与到数据缺陷统计和修复中来。事实上，人们甚至根本没有途径向任何组织报告这类缺陷。

9. **外部环境引发的缺陷** 这类缺陷起初并不是缺陷，但是随着外部条件的变化（如新

税法的实施、退休金计划的变更或其他任何可能导致软件代码变化的政府条令), 它便成为了一个缺陷。一个例子就是, 当州营业税的税点从 6% 提高到 7.5% 时, 很多的软件应用都被迫要进行更改。如果任何软件不进行更改, 那最终这款软件就会出现一个缺陷, 即使在这种变化发生前这款软件已经成功运行了若干年。由于这种变化通常是基于政府行为的, 因此这种变化虽然可能很频繁, 但却完全无法预测。

10. 修复失败 在所有试图修复软件代码缺陷的行为中, 大约有 7% 的行为会在无意中创造一个新的缺陷。有时候对软件缺陷的修复会产生次生缺陷甚至是三级次生缺陷。曾经有一个对软件提供商的诉讼案件提到, 软件提供商对一个财务软件中的缺陷连续进行了 4 次修复, 不仅没有修复原始缺陷, 还为软件带来了新缺陷。直到第 5 次修复才终于解决了问题。

11. 遗留缺陷 这类缺陷是指尽管今天才出现问题, 但其根源已经隐藏在软件中长达 10 年甚至更久的时间。遗留缺陷的一个例子就是某个无法正确计算加班工资的薪酬系统。由于加班工资已经超过了每小时 10 美元, 而记录每小时工资这个数值的输入框最大只能接受 9.99 美元。事实上, 当这个问题第一次出现并被确认的时候, 它已经存在了超过 10 年的时间(在这个问题出现的时候, 这个软件的初始开发者甚至已经不再受雇于这家公司了)。

12. 可重用代码的缺陷 软件应用中有 15% ~ 50% 的部分是基于重用代码的, 可能是购买的商业代码或者从其他软件中得到的代码。由于对这些可重用代码缺乏认证, 这些重用代码中可能包含很多缺陷和错误。但是截至 2009 年, 人们仍没有明确解决重用代码中的问题是最初开发者的责任还是重用代码使用者的责任。

13. 易出错模块 (IBM 术语) 针对 IBM 软件的研究发现, 软件错误或缺陷并不是随机分布的, 而是倾向于聚集在少数区域。例如, 在 IMS 数据库产品中, 在总计 425 个模块中, 几乎有 60% 的客户问题出现在其中的约 35 个模块中。在大型软件中, 易出错模块是非常常见的。一般来说, 大型软件系统中, 有 3% 的模块可以考虑被分类为易出错模块。

14. 事故 事故是指软件应用由于未知原因而意外停止运行。然而当我们重新启动软件之后, 它会再次正常运行。事故并不罕见, 但我们却很难去定位它的起因。可能是由于瞬间的电力波动或是电力供应中断; 也可能是由于硬件问题, 甚至是宇宙射线的影响; 又或者是由于软件缺陷所引起。由于事故是随机出现的而且几乎不能重现, 我们很难对其进行研究。

15. 安全漏洞 软件应用中有一些代码片段是病毒、蠕虫以及黑客用来获取软件权限的常用途径。日常错误处理程序和缓存溢出是安全漏洞的常见例子。截至 2009 年, 人们通常不会将这些问题定义为软件缺陷, 因为它们为恶意攻击仅有的途径因而可以进行防范。然而, 考虑到这类攻击的增长速度令人震惊, 也许我们需要重新考虑如何定义安全漏洞。

16. 心怀不轨的软件工程师 偶尔软件工程师会对他们的同事、他们的经理或是他们所工作的公司产生不满的情绪。当这种情况发生的时候, 一些软件工程师会蓄意在他们所开发的软件中植入恶意代码。这种情形最可能发生的时刻, 就是一个软件工程师在收到解雇通知之后即马上要离开公司的那段时间。尽管只有一小部分软件工程师会进行蓄意破坏, 随着经济危机越来越严重, 这种情形可能会变得越来越普遍。不管怎样, 软件工程师可以

进行蓄意破坏行动这一事实正是美国国税局会对软件工程师的纳税申报单进行手工仔细审查的原因之一。当然,软件工程师能做的不只是植入恶意代码,他们还可能使用其他方式来进行恶意破坏,如将项目经费转移到个人账户上。

17. 潜在缺陷 这个术语源于 IBM。它是在 1973 年左右被提出来的。在我所有的主要著作中都会提及这个术语。它的含义是指软件开发过程中所有可能遇到的缺陷的总数量。这个总数量包含以下 5 种缺陷来源:(1)需求缺陷;(2)设计缺陷;(3)编码缺陷;(4)文档缺陷;(5)修复失败或次生缺陷。目前,美国国内软件项目的平均潜在缺陷大约是每个功能点 5 个。预测潜在缺陷的一个经验法则是使用软件应用功能点数量的 1.25 次幂作为估算值。这种方法为规模在大约 100 个功能点到 10 000 个功能点之间的软件中可能包含的缺陷数量提供了一个较为实用的估算数据。

18. 缺陷去除效率 这个术语同样源于 IBM 并且也是在 1973 年左右被提出来的。它是指(某种测试)所发现缺陷的数量与总缺陷数量的比值。如果一个软件总共有 100 个缺陷而单元测试发现了其中 30 个缺陷,那么其缺陷去除效率就是 30%。大多数测试方式的缺陷去除效率都低于 50%,而静态分析和正式审查的缺陷去除效率最高可以达到 80%。

19. 累计缺陷去除效率 这个术语同样源于 IBM 并且也是在 1973 年左右被提出来的。它是指通过各种手段所去除缺陷的总数量,可能包括各种形式的审查、静态分析以及测试。如果通过需求审查、设计审查、代码审查、静态分析、单元测试、新功能测试、回归测试、性能测试以及系统测试等一系列的缺陷去除操作找到了约 1000 个潜在缺陷中的 950 个,那么累计缺陷去除效率就是 95%。目前,美国国内的平均累计缺陷去除效率大约仅有 85%。累计缺陷去除效率的计算是在一个固定的时间点进行,通常是在软件交付给客户之后 90 天的时候。

20. 性能问题 一些软件对于性能有着严格的标准。例如爱国者导弹的自动跟踪系统,汽车防抱死制动系统中的内置软件。如果这样的软件不能达到所要求的性能目标,那么这个软件可能是完全无法使用的或者十分危险的。然而,人们通常不会将性能问题定义为缺陷,因为软件中并没有任何不正确的代码,需要改正的是软件的执行路径。这些执行路径可能太长,或包含了过多的调用和分支。但即使这些问题不会导致明显的错误,我们仍然有很大责任去改善性能问题。

21. 循环复杂度和基本复杂度 这两个概念都涉及数学表达式。这些表达式是为判断代码片段的复杂度提供数量上的基准。这种计算方法由 Tom McCabe 博士发明,因此有时被称作“McCabe 复杂度算法”。这种复杂度的计算方法基于图论,其概括性的计算公式是“边数-节点数+2”。从实际应用的角度说,如果软件工程师在检查一段代码的时候,其循环复杂度低于 10,那么表明这段代码的复杂度较低,而如果循环复杂度高于 20,则表明这段代码非常复杂。这种计算方法非常有效,因为在循环复杂度数值和软件缺陷密度方面存在一定的关系。基本复杂度的计算与此类似,只是在计算时通过数学方法去除了多余的或重复路径的计算。

22. 有毒需求 这是一个 2009 年才出现的新术语,其来源是一个财务术语“有毒资产”(Toxic Assets)。有毒需求是指那些明确地定义在用户需求中的条目,但实际上对产品是有

危害的,如果不移除这些需求,便会引起严重的损失。不幸的是,这些有毒需求无法通过各种常规的测试进行移除,因为一旦这些有毒需求进入到需求文档中,进而进入到设计文档,那么任何基于这些文档的测试用例只能肯定这些需求的存在而不能识别出它对软件的危害。然而,正式的需求审查可以帮助我们去除掉这些有毒需求。一个著名的有毒需求的例子就是千年虫问题(Y2K),这个问题是由一个特别的用户需求所引起的。另一个更加近期的有毒需求的例子就是快速金融系统软件,如果系统的备份文件会被打开而不是存储在系统中,那么这些“快速文件”中数据的准确性就无从保证。

软件缺陷部分小结及相关结论

正如我们在本书前面部分所讨论的,当前美国国内软件业平均的缺陷数量大约是每个功能点5个。这个数量包括了各种类型的缺陷,如需求缺陷、设计缺陷、代码缺陷、文档缺陷、修复失败以及次生缺陷等。

各种缺陷去除方法累计起来只能去除掉大约85%的缺陷,因此,通常情况下软件应用在交付时每个功能点大约包含0.75个缺陷。需要注意的是,在产品交付时所有早期缺陷(包括需求缺陷和设计缺陷)都已经进入到了产品代码中。换种方式来讲,尽管千年虫问题源自于设计缺陷,但最终这个缺陷还是进入到了产品代码内。没有任何一种编程语言是对其免疫的,因此对于所有使用已知语言开发的数以千计的软件应用来说,千年虫问题是普遍存在的。

对于一个包含了1000个功能点的典型软件应用来说,每个功能点0.75个缺陷意味着发布出去的软件中包含了大约750个缺陷。在这些缺陷中,大约有20%是严重等级较高的缺陷,也就是说当用户拿到软件的初始版本的时候,里面可能包含了约150个非常严重的缺陷。

我们可以采取以下5类补救措施来改善这种状况:

1. 对软件机构所发现缺陷进行100%的评估。
2. 对所使用的所有审查方法、静态分析方法及测试方式进行缺陷去除效率评估。
3. 使用多种有效的缺陷预防方式,如联合应用设计(JAD)、质量功能展开(QFD)等,以减少潜在的软件缺陷。
4. 使用多种方式的正式审查、静态分析以及改进测试方法来提高缺陷去除效率。
5. 检查缺陷去除费用、总开发费用以及总开发周期和维护费用对于提高质量的结果。

以上5种主要的措施结合起来,可以将潜在的缺陷数量降低到每个功能点3个以下,同时可以将缺陷去除平均效率提高到95%,对于一些关键性的软件应用,甚至可以达到99%。

软件工业应该努力将平均缺陷数量降低到每个功能点3个以下,将缺陷去除效率提高到95%以上并将所交付产品中所包含的缺陷数量降低到每个功能点0.15个以下,这是一个可以实现的目标。

将更准确的评估、更有效的缺陷预防以及更高的缺陷去除效率结合起来,我们可以将一个1000个功能点的交付物中的缺陷数量从750个降低到150个。在这150个缺陷中,只有10%左右是严重等级比较高的缺陷。因此,与现今通常情况下会包含150个高严重等级

缺陷相比，我们很可能只会发现 15 个高严重等级缺陷。这是整整一个数量级的进步。

更好的消息是，观察数据显示，那些质量等级较高的软件产品通常开发周期更短、开发费用更低并且维护费用也较低。

实际上，软件开发周期变更和费用超支的主要原因是在测试初期出现了大量缺陷。大部分项目的进度在测试开始之前都是按计划进行的，费用也在预算之内，但是测试阶段总会出现大量缺陷使得测试周期和费用在原计划的基础上增加了数倍。

在 2009 年的今天，实现更好质量目标的技术事实上已经存在了，只是还没有得到广泛应用。这就意味着，在 2009 年前后，软件工业的主要弱点，在于亟待提高的质量意识和对于质量的经济学价值的认识。

8.11 软件源代码缺陷的预防和去除

在软件开发阶段的平均缺陷数量大约是每个功能点 1.75 个，对于那些代码行和功能点比例大约是 100 的开发语言来说，就是每千行代码 17.5 个缺陷。正如本书前面所指出的，缺陷数量会随着编程语言的不同而变化，同时，缺陷数量也会因为开发团队的经验和技巧的差异而有所变化。

假设我们使用同一种编程语言，软件源代码中的缺陷可能低至大约每个功能点 0.5 个或者说每千行代码 5 个缺陷，也可能高达每个功能点 3.5 个或者说每千行代码 35 个缺陷。

但是，即使我们不考虑其潜在缺陷的范围是非常宽泛的这一因素，代码中的缺陷仍然比其他任何种类的缺陷数量都要多。针对代码缺陷的去除方法可以除去 80% ~ 99% 的缺陷，当然如果能达到 99% 的去除效率肯定比 80% 的去除效率要好，但即使是在最好的情况下（即去除效率可以达到 99% 的情况），也仍然会有一些缺陷被遗漏掉。

当我们考虑代码缺陷以及其他各种缺陷时，我们需要通过两种渠道来提高代码质量：

1. 缺陷预防，或者其他可以降低潜在缺陷数量的方法。
2. 缺陷去除，或者其他可以找到并消除代码缺陷的方法。

目前可用的预防代码缺陷的方式包括使用经过认证的可重用代码模块、使用通用的标准编程方法或模式、使用结构化编程方法、使用高级编程语言、使用以前正式开发过程中的结构化原型、将大型软件应用分割为若干小部分（比如敏捷开发）、使用代码审查、使用基于测试的开发方式以及使用静态分析工具。通常人们认为结对编程（Pair Programming）^①对于软件质量的提高也会有一定效果，但是实际项目中很少使用这种方法因而我们并没有充分的数据来支持这一说法。

目前可用的代码缺陷去除方式包括手工检查、结对编程、使用调试工具、代码审查、

① 结对编程是一种敏捷软件开发的方法，两个程序员在一个计算机上共同工作。一个人输入代码，而另一个人审查他输入的每一行代码。输入代码的人称作驾驶员，审查代码的人称作观察员（或导航员）。两个程序员经常互换角色。在结对编程中，观察员同时考虑工作的战略性方向，提出改进的意见，或将来可能出现的问题以便处理。这样使得驾驶员可以集中全部注意力在完成当前任务的“战术”方面。观察员当作安全网和指南。结对编程对开发程序有很多好处。比如增加纪律性，写出更好的代码等。——译者注

静态分析工具、17种传统测试方式以及自动化单元测试和回归测试。

我们很难去研究那些由单个软件工程师独立完成的缺陷去除工作。手工检查、调试以及单元测试通常情况下由软件工程师独立完成。这更加像是一个人的行为，没有人进行监督，也没有详细测试记录保存下来。大多数企业的缺陷跟踪系统直到公开的缺陷去除阶段（通常包括正式审查、功能测试及回归测试等）才开始收集数据。我们通常很难去了解在这些公开测试活动开始之前发生的事情。但是，还是有一些例外的。

IBM曾一度召集志愿者记录那些由他们在自己所写代码中发现的缺陷，这项研究的目的是为了发现这些缺陷去除方法的实际缺陷去除效率，因为通常人们是不了解这些方法的。显然，这些数据并不是用来进行任何惩罚的，并且也没有产生任何统计报告，而是作为机密信息保存起来。

再往里说，由Watts Humphrey发明的个体软件过程（Personal Software Process, PSP）和团队软件过程（Team Software Process, TSP）都包括了整个代码开发过程的缺陷记录。

遗憾的是，敏捷开发完全走向了另外一个方向，在敏捷开发过程中通常不会去记录那些个人进行的缺陷去除行为。事实上，很多敏捷开发项目甚至根本不去记录任何缺陷数据。这显然是一个错误，因为这样敏捷开发方法便会缺少其在提高软件质量方面的数据，也就降低了证明敏捷开发在这方面价值的能力。

本书的第9章将会讨论那些公开的缺陷去除方式并讨论它们在质量方面的影响。本章中，我们会着重介绍那些个人独立完成的缺陷去除方式，因为其他软件工程文献很少会涉及这方面的内容。

个人缺陷去除方法并不像那些公开的缺陷去除方法（如正式审查、静态分析以及引入了测试专家或者质量保证人员的各种测试阶段等）那样有大量的关联数据可以进行研究分析。但是为了保持内容的完整性，我们需要包含个人缺陷预防方法和个人缺陷去除方法。

在讨论个人缺陷预防方法或个人缺陷去除方法的效率之前，我们有必要注意一点，那就是不同的软件工程师或编程人员在经验和技巧方面可能会有很大的不同。

IBM曾经进行过一次对照实验，在这项实验中，数名程序员要完成同样的一个实验产品样例，在最终结果中，对于同一个功能特性，最庞大的解决方法和最简洁的解决方案在代码量上面的比例大约是6:1。

类似的研究也向我们证实，不同程序员为同一个问题进行编码和调试的时间可能相差10倍左右。

软件工程师在个体能力上的巨大差异意味着在我们所进行的研究中，众多软件工程师个体差异可能比我们所研究的方法、工具或其他因素对研究结果的影响更大。

8.12 编程缺陷预防方法

衡量缺陷预防效果或确定缺陷预防方法所消除的缺陷数量比衡量缺陷去除效果要困难得多。对于缺陷去除来说，我们通常可以逐渐收集到通过使用这些方法所发现的缺陷数量以及对应的严重等级。

当软件产品交付给客户之后，我们会继续统计缺陷数量。在大约交付使用了90天之后，我们可以将软件团队内部所发现的缺陷数量和客户所发现的缺陷数量进行对比，从而计算出缺陷去除效率。比如，产品开发过程中软件团队发现了85个缺陷，而客户在使用过程中发现了15个缺陷，那么缺陷去除效率就是85%。这样的数据既有价值，也比较容易进行收集，同时也是相当准确的，除了没有考虑那些通过个人缺陷去除方法如手工检查或单元测试所发现的缺陷。

但是对于缺陷预防来说，我们很难找到一个方法去测量其中的缺陷数量。如果要对缺陷预防效果进行研究的话，目前可行的方法是需要收集大量软件项目的数据。在这些项目中，一部分项目使用了某种缺陷去除方法而另一部分项目没有使用这种方法。

例如，假设我们对100个项目的样本进行分析，其中50个项目使用了结构化编程方法而另外50个项目没有使用结构化编程。假设在这50个使用了结构化编程的项目中，其平均缺陷数量是每千行代码10个编程缺陷或者说每个功能点1个缺陷；而在另外50个没有使用结构化编程的项目中，平均缺陷数量是每千行代码20个缺陷或者说每个功能点2个缺陷。通过以上分析，我们可以做出一个假设，那就是结构化编程方法可以预防50%代码缺陷的出现，但是这仍然仅仅是一个假设，并不是确实的证据。

更进一步说，现实中的情况从来都不是简单的，因而很难进行全面考虑。在实践中，不同的项目在执行时可能会有很多的不同因素，如是否使用了静态分析、是否使用了高级编程语言、是否使用了审查、其项目团队在开发经验上面的差异，又或者所解决问题复杂度的不同等诸多因素。

众多因素同时影响缺陷预防的效率，这意味着其中任何一项因素的准确去除效率只能是一定程度上的主观数据，并且很有可能会一直持续如此。

学术研究机构可以组织学生进行对比实验，这些实验仅就众多因素中某一个因素的效率进行研究，但是这种对比实验极少对缺陷预防进行研究。

尽管如此，通过对于数百名软件工程师以及数百个软件项目进行多年的长期大范围观察，我们有一些比较合理的数据可以支持我们对缺陷预防的一些因素进行客观研究和分析。

1. 代码重用

如果可用的重用代码拥有零缺陷等级的认证，或者至少在成为可重用代码之前经过了严格的审查、测试以及静态分析，那么使用这些可重用代码是众所周知的最好的缺陷预防方式。如果说在使用这些可重用代码进行定制开发的时候，其平均缺陷数量是每千行代码15个的话，那么只有极少一部分缺陷是来自这些可重用代码自身的，有时候只有百分之二的比例。

然而，有一个非常重要的事情，那就是使用未经认证的重用代码不仅非常冒险，还会导致高昂的开发费用。如果这些未经认证的可重用代码所包含的缺陷数量超过每千行代码1个的话，而同时又有超过10个软件应用使用了这些代码，则当这些应用被整合到一起时，所需要的调试费用会非常高昂，以至于这个软件项目会变成负收益。

尽管在所有的缺陷预防方法中，使用认证的可重用代码是最有效的一个，因而通常会被认为是最佳实践，但同时它也是最少应用的方法。未经认证的重用代码远比经过认证的

重用代码要多，其比例至少是 50 : 1。人们可以将使用认证的可重用代码及其相关材料定义为是最佳实践，但使用未经认证的重用材料必须要定义为危险的实践。

对于软件工程师来说，调试别人所写的不熟悉的代码远比调试自己的代码要困难得多。如果可重用的代码中包含任何错误的话，每一次我们一个新的软件中使用这段代码时，相同的错误很可能会一再出现。因此，使用未经认证的代码是十分危险的，其最终所需要的开发费用甚至可能比完全不使用这段代码而独自进行开发所需要的费用还要高。这就是为什么使用未认证代码的项目会出现负收益。

可重用代码的来源很多，包括商业软件供应商、遗留应用、面向对象类的库、公司可重用组件库、公共领域以及开源软件库等。尽管可重用代码的资源相当丰富，但相比之下，对这些可重用资源进行维护的数据就不那么丰富了（更多信息可以参考本书前面关于可重用材料认证的部分）。

如本书中其他部分所提到的，在可重用组件的世界中，重用代码只是其中的一部分。其他可重用的组件还包括可重用的设计、数据结构、测试用例、使用说明书、工作分解结构（WBS）以及帮助文档。这些材料应该和它们所支持的代码整合在一起，打包提供给使用这些组件的人们。

2. 编码模式

参与过众多软件项目的编程人员或软件工程师应该会注意到，在许多软件应用中，某些代码序列会出现很多次。这些代码序列可能是执行对输入数据进行验证的功能，以保证输入数据出现错误时程序能够正确处理。如拒绝数字输入框内的字符数据，或者确保文本和数字数据中所包含的字符数不会超过程序设计中指定的限制。

这些代码序列就是编码模式，它们通常是通过个人的经验所得到的，尽管这些代码是个人的且未经过正式认证，但它们是可以重用的。然而，考虑到类似内容非常多，我们可以将这些编码模式记录下来并用图片的方式来原因，然后用来对刚进到这一行业中的新软件工程师们进行培训。这已经成为一个非常明显的事实。

基于编码模式的开发方法对于缺乏经验的新软件工程师非常有帮助，这种方法有潜力将这些新开发者所开发代码中的潜在缺陷降低 50% 以上。一旦人们大规模发布出多种标准模式供开发者们使用，这些模式还可以帮助开发者们将职业由某一类软件开发转向另一类软件开发。例如，同嵌入式软件开发有关的模式和同信息技术软件相关的模式都有很多不同的种类。

在 2009 年左右的时候，基于编码模式的开发方法所缺少的，是一个对于各种可用模式的分类系统，这个分类系统可用于对模式进行分类登记并帮助人们选择适合的模式组合。同时，人们对于目前有多少种有价值的可用模式并没有一个准确的了解。在未来，毫无疑问使用模式进行开发会被认为是一种最佳实践，尽管在 2009 年这样做可能是早了几年。

对于那些工作范围仅仅专注于一小类领域的软件工程师来说，他们可能仅仅会使用 25 种到 50 种通用模式，主要集中于输入输出数据验证、错误处理或者有关于安全的问题。但是如果我们要考虑的包括所有种类以及各种形式的软件，如财务软件、嵌入式软件、网络应用、操作系统、编译器、航空电子软件等，那么有价值的软件模式的总数量可以很容易

超过 1000 种。这个数字太过庞大，因而人们很难在其中随机抽取到特定的模式。因此，如果希望这些模式想要成为行业中更有价值的工具的话，我们需要对这些模式进行组织分类。

3. 审查

正式审查对于缺陷预防和缺陷去除都同样有效。正式审查的参与者会自觉地避免在自己的工作中出现审查中所发现的问题。因此，在参加了多次审查之后，编码缺陷数量相比于使用审查之前，大约会减少 80% 以上。由此可见，正式审查可以作为两方面的最佳实践：既是高效的缺陷预防方法，也是十分有用的缺陷去除方法。

审查在缺陷预防方面有着十分显著的效果，在实施审查一年之后，审查对于参与者来说变得越来越枯燥，因为很难再发现一些有趣的缺陷或问题了。但不幸的是，许多公司因而停止了使用审查，但在此之后软件缺陷的数量又开始不断增长。

审查还有另外一个非常有价值的作用，那就是当初学者审查专家们的工作时，他们会自然地学习到一些技巧来提高编程技术；而当专家们审查初学者的工作时，他们可以对初学者提出一些非常有用的建议，同时也会发现这些初学者工作中的很多缺陷和问题。因此，参与审查者中包括一些专家或者顶尖软件工程师是非常有用的。

4. 自动化静态分析

静态分析是一种不同于测试的新技术。自动化静态分析工具有其内置的规则和逻辑，这些规则和逻辑的设定是为了能够在软件代码中发现各种常见形式的缺陷。这些工具通常都非常有效，其缺陷去除效率可以达到 85% 以上。但是需要说明的是，这些工具仅仅支持现有的 2500 种语言中的大约 50 种。在这 50 种支持的语言中，主要都是现代语言如 C、C#、C++、Java 等类似语言。一些比较古老的语言或者不是非常出名的语言，如 MUMPS、Coral、Chill 等类似语言是不支持的。然而，在我们现有的约 100 种静态分析工具中，有一些可以支持较为古老的语言或者某些专门化语言，如 ABAP、Ada、COBOL 及 PL/I。有些工具还提供了可扩展的规则，因此在理论上讲，现存所有的 2500 种语言也许都可以使用静态分析工具，虽然这种情况不太可能真的出现。

由于静态分析工具对于发现软件代码中的缺陷非常有效，而通常情况下是由编程人员来运行静态分析工具的，这些工具也可以用来作为缺陷预防工具从而为提高软件质量带来益处。换句话说，如果一个编程人员对于自动化静态分析工具所发现的缺陷进行认真分析和处理的话，他就会自觉地在将来的工作中避免出现相同的错误。

截至 2009 年，人们将使用静态分析工具定义为其所支持语言中预防缺陷的最佳实践。静态分析在缺陷去除方面的作用有着十分明显的证据，同时，其在缺陷预防方面所作贡献的证据也在不断增加中。

静态分析工具在开源软件开发社区中有着广泛的应用，并且都取得了不错的结果。考虑到静态分析的能力和作用，其使用范围正在扩大，而我们认为这种方法应该成为软件开发过程中的一个标准过程。事实上，静态分析工具应该作为每一个软件开发环境和软件维护环境的一部分，而使用静态分析方法也应该成为所有软件开发和维护过程的一部分。

5. 基于测试的开发方法

极限编程方法 (Extreme Programming, XP) 中包含了这样的一个概念, 那就是在开发软件代码之前进行测试用例的开发。实际上, 在这种方法中, 测试用例是用来帮助软件团队进行需求收集和软件设计的。

这种先开发测试用例的方法主要致力于软件质量的提高, 因此基于测试的开发方法 (Test-based Development, TBD) 在缺陷预防和缺陷去除方面都被认为是一个最佳实践方法。但因为 TBD 方法还是一个新出现的方法, 我们目前还没有基于众多实验项目的观察数据来证明它的效率。同时, 敏捷开发项目对于项目数据测量的不严格也加剧了我们收集 TBD 方法实际效率的难度。

但是, 根据一些传闻所提供的证据, TBD 方法可能可以降低 30% 的潜在缺陷数量, 同时将单元测试的缺陷去除效率从 35% 左右提高到约 50%。这两项结果的调查方向都是正确的, 但是我们仍然需要更多的数据来证明。目前, TBD 方法非常有潜力成为一个最佳实践方法, 一旦人们可以得到更多的定量观察数据, 毫无疑问 TBD 方法会作为最佳实践方法得到认可。

6. 高级编程语言

在高级语言诞生之初, 其开发者所声明的优点之一就是高级编程语言可以减少潜在缺陷的数量, 与此相关的另外一项声明说, 如果缺陷真的存在, 人们会更容易在高级编程语言所开发的软件中发现这些缺陷。这两项声明看起来都是有根据的, 但实际情况要复杂得多, 同时我们在讨论高级语言的效率时, 总有一些通用规则之外的例外情况。

很明显, 如果软件源代码的数量减少, 那么错误出现的机会也会减少。假设完成某一个特定的功能, 如果使用汇编语言, 需要 1000 行代码, 而使用 Java 语言的话仅需要 150 行代码, 则大概率上 Java 代码中的缺陷会更少。即使两种代码所编写软件中所包含缺陷的比例一样, 如每千行代码 10 个错误, 那么较为庞大的汇编语言程序可能会有 10 个缺陷而同时较小的 Java 程序可能只有 1 ~ 2 个缺陷。

但是, 有些高级编程语言的语法结构相当复杂, 因而便增加了无意中在代码中出现错误的可能性。APL 语言便是这样的一个例子, 这门语言是一门相当高级的语言, 但是其语法结构也很复杂以至于人们很难去阅读并理解代码, 因而也很难去进行调试, 特别是在试图去调试其他人所编写的程序的时候。

观察表明, 那些使用常见语法结构、使用了各种助记标签同时使用了人们易于理解和掌握的命令的编程语言, 相比那些等级相同但使用了非常晦涩难懂的命令和复杂命令 (如包含了很多层嵌套的命令) 的编程语言, 前者在一定程度上可能导致的软件缺陷更少。

如果学术研究机构能够就不同编程语言在解决标准问题时的缺陷比例和调试时间进行对照实验的话, 其结果将会非常有趣, 同时也是非常有用的信息。人们会非常有兴趣去查看各种流行的编程语言的缺陷数量和调试时间, 如 C、C #、C++、Objective C、Java、JavaScript、Lua、Ruby、Visual Basic 及其他 50 种语言。然而, 截至 2009 年, 似乎并没有人曾经进行过类似的对照实验。

截至 2009 年, 编程语言的过剩以及因此而导致的对于软件维护费用的负面影响, 使得

人们怀疑使用任何一种特定的编程语言是否为最佳实践方法。

7. 原型开发

对于大型的复杂软件来说，软件工程师们可能需要尝试若干种不同的代码序列之后，再从备选方案中挑选出一个最好的方案应用在软件的最终版本上。在这种模式中，使用原型进行开发对于减少最终版本中的缺陷是非常有用的，因为它可以帮助软件工程师通过一种较为良性的方式去尝试各种备选方案。

通常情况下，软件团队只针对软件开发中最为复杂和棘手的部分开发原型。因此，通常情况下原型的规模大约只占整个软件应用规模的 5% ~ 10%。这种只集中于最棘手问题的方式使得原型非常有效，同时原型规模的紧凑和简洁也防止了其开发费用的过度使用。

软件应用的原型通常有两类：一次性的和逐渐进化的。顾名思义，一次性的原型用来对各种运算法则和代码序列进行试验，之后就会被丢弃；而另一方面，一个逐渐进化的原型会逐步地发展并成为最终产品的一部分。

由于原型通常是为了进行试验而快速开发出来的，在某种程度上来讲，一次性的原型比逐渐进化的原型要安全。由于快速开发的原因，相比软件开发中其他成熟的工作，原型中可能会包含更多错误或缺陷。如果我们试图将原型转化为最终产品的一部分，那么很可能会导致最终产品中出现过度的缺陷。

使用一次性原型来尝试不同解决方案，或对于较为困难的编程问题进行尝试，这种方法应该成为一个最佳实践。但是，对于那些未经过严格仔细开发而仅仅是为了加快开发速度的逐渐进化类原型，在最终产品中使用它们不能算是一个最佳实践，相反，这是一个十分冒险的行为。

8. 结构化代码

1968 年 8 月的《ACM 通信》上面发表了一封 Edsger Dijkstra 教授写给编辑的信，这封信名为《Go to 语句有害论》的信是软件工程历史上最为著名的信件之一。

这封信的主要论点就是软件代码分支的过度使用或者说 Go to 语句的滥用使得软件应用的代码结构变得非常复杂，人们在编写代码过程中可能会出现由于错误的分支语言序列而导致缺陷，同时这种缺陷也很难发现和去除。

这封信引发了软件编程风格的一次革命，使得人们开始了解“结构化编程”的概念。在结构化编程原则的指导下，软件代码中的分支开始减少，编程人员逐渐意识到各种复杂的分支嵌套结构和那些自作聪明的代码序列会引发缺陷，同时会使得人们很难去测试和检验产品代码。

巧合的是，另一位软件工程行业的前驱者，Tom McCabe 博士，在 1976 年 12 月的《IEEE Transaction on Software》上面发表了一篇文章，阐述了他所发明的一种对代码结构的测量方法。McCabe 博士提出的这种测量方法提到了这样的两个概念：循环复杂度和基本复杂度。

循环复杂度的计算是以图论为基础的，是一种正式的衡量软件应用控制流程图复杂性的方法。通过计算控制流图中边的数量以及节点的数量，我们可以使用如下公式计算循环复杂度：循环复杂度 = 边数 - 节点数 + 2。

基本复杂度的计算同样是以图论为基础，只是在计算时去除了代码中多余的或重复路径的计算。

我们来看循环复杂度的计算，根据公式，一个没有任何分支语句的代码段，其循环复杂度为1。这表明这段代码的执行是完全线性的，没有任何分支或者 go to 语句。从心理学的观点来说，人们通常认为循环复杂度低于10的代码有着良好的结构。但如果一段代码的循环复杂度超过20，通常人们会很难理解这段代码，同时，这段代码很难不出任何错误地从头到尾执行完成。

有一些观察数据表明，那些循环复杂度低于10的代码段中的缺陷数量，大约只有那些循环复杂度高于20的代码段中缺陷数量的40%。如果我们将其他因素（如编程语言或程序员的经验水平）保持一致的话，那些循环复杂度为1的代码应该是缺陷最少的。

IBM 曾经进行过一次调查并发现了一个惊人的结果：在同样规模的代码中，有时候资深编程人员或者有经验的编程人员的代码缺陷数量比那些编程新手还要高。然而，这种反常现象的实际原因是，通常编程专家从事的都是非常复杂和困难的软件开发，而新手通常只开发一些非常容易理解的简单程序。无论如何，这项研究表明，问题复杂度对于代码中的缺陷比重有着十分显著的影响。

随着循环复杂度和基本复杂度对于代码缺陷的重要性的增加，人们开发出了若干种商业工具来计算软件复杂度。在2009年左右，市场上有很多可用的工具可以对多种编程语言代码的循环复杂度和基本复杂度进行计算。

在20世纪80年代，市场上有一些针对 COBOL 语言的工具。它们不仅能够计算代码的复杂度，同时还可以自动地重构代码以便降低其循环复杂度和基本复杂度。这些工具声称（这些声明作为证据被保存了下来），重构后的代码有较低的复杂度，而且在进行修改和维护时，它们比原始代码需要的人力要少。

使用结构化编程技术和保持较低的复杂度水准都可以视作最佳实践。低复杂度、分支少的代码通常会包含较少的缺陷，而且其中的缺陷通常也较容易发现。因此，结构化编程方法应该算做缺陷预防的最佳实践之一。

9. 段落分割

超过50年的观察数据可以确凿地证明，潜在缺陷的数量和软件的规模几乎是完全成正比的，无论软件的规模是用代码行还是功能点来计算。既然规模和缺陷是紧密联系在一起的，我们很有理由去问这样一个问题：我们为什么不将大型软件系统分解为若干个小片段呢？

不幸的是，这件事情并不像听起来那么容易。我们来做一个对比，既然大家都知道建造一艘80 000吨的大型游轮所需费用十分高昂，而小型船只的造价却要便宜得多，我们为什么不将这艘船分解为80 000个小型船只呢？很显然，80 000个小型船只所提供的功能和所满足的用户需求和一艘80 000吨大型游轮的功能和需求是完全不同的。

截至2009年，人们仍然没有找到一种行之有效的方法来成功地将一个大型软件系统分割或分解成为若干独立的小型组件。凑巧的是，敏捷开发方法成功地将一个软件系统分解为若干部分或者说 sprint，之后再按次序对每一个部分进行开发。但是大多数敏捷开发的软件应用在规模上都小于10 000个功能点，在架构上也相对简单。

目前还没有任何一个敏捷开发项目处理过规模类似于 Microsoft Vista (大约 150 000 个功能点) 或大型 ERP 软件 (大约 300 000 个功能点) 的产品。事实上, 如果使用敏捷开发模式来开发这种规模的产品并且保持敏捷开发团队平均规模 (少于 10 人) 的话, Vista 大约需要 150 个 sprint 才能开发完成, 而 ERP 软件则需要 300 个 sprint。假设一个 sprint 持续一个月, 则 Vista 的开发周期大约是 12 年而 ERP 的开发会持续 25 年。多个敏捷开发团队可以加快开发速度, 但是处理不同团队之间的代码接口无疑会增加工作的复杂程度, 同时也会使缺陷增多。

概括来讲, 如果我们可以完美地将大型软件分割为独立的小型软件包或者小型组件, 那么这种开发方法将会非常有效。但是考虑到许多大型软件系统的功能组合及其架构设计, 这种分割并不是总能实现。因此截至 2009 年, 考虑到我们还缺少分割软件的标准和有效方法, 我们还不能将这种分割开发方式定义为最佳实践。

对于大型软件来说, 对主要功能进行分割是很普遍的, 但是每个分割后的功能组件自身通常包含 10 000 个功能点以上。目前仍然没有任何行之有效的方法可以将一个包含了 150 000 个功能点或 1500 万行代码的大型软件系统分解为约 15 000 个独立的小部分。在 2009 年左右, 最好的情况大约是将这种大型系统分解为 10 个左右的大型软件片段。

10. 缺陷评估和记录方法

Watts Humphrey 所开发的个体软件过程 (PSP) 和团队软件过程 (TSP) 重点强调了仔细记录软件开发过程中的所有缺陷。这些缺陷包括了那些通过个人缺陷去除方法如手工检查、单元测试所发现的缺陷, 通常情况下这些缺陷对外是不可见的。

记录特定缺陷的目的是为了将这些缺陷留在软件工程师或者编程人员的记忆中。通常情况下, 经过在一系列项目中实施这种方法后, 代码中的缺陷大约会下降 40%, 因为人们会自觉地避免去犯相同的错误。

因此, 在缺陷预防方面, 缺陷评估和记录是非常有用的, 因为人们会倾向于将注意力放在缺陷上面, 随着时间的推移, 人们会开始尽力去避免和减少缺陷。记录缺陷并专注于质量的方法可以定义为一个最佳实践方法。

团队软件过程 (TSP) 还有一个很不寻常的现象, 其在缺陷预防上的效果似乎是随着软件规模的增加而增加的。换句话说, TSP 在应用于超过 10 000 个功能点的大型软件系统时会更加成功。这种情况在众多开发方法中是极少发生的。

11. 结对编程

结对编程的概念是指两个软件工程师或编程人员共用一个工作机。他们轮流进行代码编写, 在一个人编写代码时, 另一个人观察所编写的代码并提出意见和建议。结对编程人员还可以在开始对模块或软件片段进行编码前讨论各种备选方案。

有一些实验数据表明, 结对编程方法在缺陷预防和缺陷去除方面可能是有效的。但是, 截至 2009 年, 结对编程方法在实际软件项目中的应用还非常罕见, 我们也不太可能去评估这种方法在大型软件项目中的效果。

表面上看, 结对编程方法非常有可能需要两倍人力来开发一段指定的代码片段。实际上, 考虑到通常情况下人们更倾向于在谈话中讨论社会问题而不是软件问题, 我们有理由

怀疑结对编程的费用会比单独开发的两倍还要高。

除非我们有了可用的实际项目数据，而不仅仅是一些小型实验的数据，我们目前仍然没有足够的数据来判断结对编程方法在缺陷去除或缺陷预防方面的作用和影响。

12. 缺陷预防的其他方法

本章前面提到的缺陷预防方法都有着较为广泛的应用，因此它们对于代码缺陷预防方面的效果是可以进行推测和考察的。其他一些方法看起来会对缺陷预防有一些益处，但是我们很难对其进行判断。这样的方法之一就是应用于软件开发的六西格玛方法（Six Sigma）。六西格玛方法确实包含了对软件缺陷进行评估及缺陷原因分析的因素，然而考虑到六西格玛方法通常是应用于企业级别而不是单个特定的软件项目，我们很难去评价其效果。其他一些缺陷预防方法可能会对预防缺陷的发生有一些帮助，但是笔者手中并没有令人信服的数据来说明其效果，这些方法包括质量功能展开（QFD）、根本原因分析、Rational统一过程（RUP）以及其他各种敏捷开发方法。

13. 缺陷预防方法的联合应用与协同配合

尽管前面所提到的各种缺陷预防方法可能是独立出现的，但人们经常混合使用多种方法，有时候各种方法可以进行协同配合。例如，结构化编程通常会和团队软件过程（TSP）方法结合在一起，或者和审查及静态分析混合使用。

最常见的联合应用是高级编程语言和结构化编程方法的协同使用。同时，能够产生最高水准的缺陷预防效果的合作组合当属在软件过程方法（如团队软件过程，TSP）中使用高级编程语言、认证的可重用代码、开发模式、开发原型、静态分析以及审查方法。

对于缺陷预防和缺陷去除来讲，软件工程师的个人经验和技术水平比其他所有因素都重要，这一因素一直占据着统治性的地位。然而，截至2009年，软件工程领域仍然缺乏一个评价个人业绩和能力的标准方法，这里没有从业执照或行业认证，没有专家委员会，也没有方法来评判从业者的渎职行为。因此，对于软件工程师的评估虽然重要，但是却很难去执行。

14. 缺陷预防方法的观察与小结

由于对缺陷预防方法有效性进行测量的困难性和结果的不可靠性，一系列缺陷预防方法相比于缺陷去除方法，仍然缺少大量令人信服的统计数据。

个人的缺陷预防方法尤其难于进行研究，因为大多数此种缺陷预防行为都是个人行为因而极少有可用的记录或者统计信息，除了个别志愿者提供的信息。

通过长时间对数百个软件项目和软件工程师进行评估，其结果为我们提供了一些有利的证明来表明哪些对于缺陷预防确实是有作用的，但是这些结果仍然不够明晰，并且这种情况将可能会一直持续下去。

8.13 缺陷去除方法

目前人们有很多关于公开缺陷去除方法（如正式审查、功能测试、回归测试、独立验证和确认等）的统计数据。但是个人缺陷去除方法就完全是另外一回事了。个人缺陷去除这个

阶段是指那些由软件工程师或编程人员自己进行的缺陷去除行为，这些行为通常没有目击者同时也没有任何书面记录保存下来。

个人缺陷去除方法主要有以下形式，但以下列出来的并不是仅有的形式：

1. 手工检查
2. 使用自动化工具进行调试
3. 自动化静态分析
4. 子程序测试
5. 手工单元测试
6. 自动化单元测试

由于这些缺陷去除方法中的大多数都是个人私下使用的，能够判断其效率的数据要么来自于那些保存了所发现缺陷记录的志愿者，要么是来自那些记录所有软件缺陷方法的使用者，如个体软件过程或团队软件过程方法的使用者。

自动化静态分析的适用范围既包括那些独立编程人员用来对他们自己的代码进行缺陷去除，也包括那些公共领域的开源软件开发人员在大型软件项目（如 Firefox、Linux 等）上进行协同工作。因此，在公共领域的使用中，静态分析方法积累了大量充实的数据来证明其效率，同时，我们可以假设静态分析在个人应用中拥有同样的缺陷去除效率。

1. 手工检查

在计算机技术及编程技术发展初期，编写代码工作和对代码进行组合及编译工作之间有时候会有长达 24 小时的延迟。为了将程序源代码转换为穿孔卡片并通过一定的次序进行组装和编译，人们需要耗费数个小时的时间才能开始执行程序或对程序进行测试。

在编程技术发展初期的 19 世纪 60 年代和 70 年代，个人缺陷去除很常用的一个方法就是手工检查或者仔细阅读程序源代码以寻找其中的缺陷。同时，手工检查在技术上是必需的，因为大量的穿孔卡片中如果存在任何错误，都会使得组装或编译过程中断，人们就可能需要另外的 24 小时才能够开始进行测试工作。

在 2009 年的今天，代码片段在编写完成后可以立即进行编译或转换，也可以立即执行。事实上，人们可以在包含了调试工具和自动化静态分析工具的开发环境中立即执行这些代码。因此，随着个人工作机和个人开发环境的普及，手工检查的应用频率也在下降。

尽管没有很多的近期数据来证明手工检查的效率，30 年前的历史数据表明，手工检查在缺陷去除方面大约有 40% 到略高于 60% 的去除效率。

在 2009 年的今天，手工检查主要是为了发现那些少数棘手的缺陷或者其他缺陷去除方法难以成功发现和去除的缺陷。这种缺陷包括安全漏洞、性能问题以及偶尔进入到软件源代码中的有毒需求。静态分析或常用测试方式很难发现这类错误，因为这类错误不会导致代码中出现明显错误（如导致程序产生错误走向的分支或者违反边界规则等）。

在软件所有可能被发现的缺陷中，这类特别的缺陷大约只占百分之五的比例。对于这些其他缺陷移除方法难以发现的问题，手工检查有着接近 70% 的缺陷去除效率（手工检查的去除效率不能再高的原因，是因为很多软件工程师通常不会意识到某一种特殊的代码形式实际上是错误的。这就是为什么我们需要对文章的手稿进行校对，通常文章作者不是总

能发现自己所犯的错误)。

尽管使用正式审查方法可以找到这些不易发现的缺陷,但正式审查方法在软件项目中的应用率并没有超过 10%,而且这种方法需要 3 到 8 个人参与进来。另一方面,手工检查仅需要一个人就可以进行,既不需要任何正式的准备工作的,也不需要任何培训,而且任何时间都可以进行。

在 2009 年的今天,手工检查仅仅是作为一种附加的缺陷去除方法,通常不是所有的项目都需要使用这种方法。它对于一小部分不易发现的缺陷非常有效,因此在需要使用它的场合,手工检查方法可以视作一个最佳实践方法。

2. 自动化调试

在 2009 年左右,软件工程师和编程人员可以使用数百种调试工具。这些工具通常会支持某些特定的编程语言,如 Java 或 Ruby,或者支持特定的操作系统,如 Linux、Leopard、Windows Vista 等。不管哪种情况,人们可以使用的调试工具数量繁多、形形色色。

调试工具的功能各不相同,但是所有的工具都允许在执行代码时在不同的地方设置断点;允许对代码进行修改;还可能包括在代码中寻找一些常见问题(如缓存溢出或分支错误等)的功能。除此之外,为某种语言或者操作系统特别定制的调试器可能还会有一些和这种语言或操作系统相关的特殊功能。

调试工具是一种常用的工具,因此使用这种工具已经成为了行业内的普遍惯例。因此,使用调试工具可以定义为一种最佳实践方法。也就是说,没有任何一种调试工具是百分之百有效的,人们仍然可能会漏掉一部分缺陷。事实上,考虑到调试后的代码在审查、静态分析及测试中所出现的缺陷数量,程序调试器的平均缺陷去除效率大约只有 30%,甚至更低。

3. 自动化静态分析

静态分析工具通过检查源代码及源代码的路径来寻找常见的错误。这些工具中,一部分使用内置的一系列规则进行工作,另外一部分工具则允许用户对规则进行定制和扩展。

在网络上使用“自动化静态分析”(automated static analysis)作为关键词进行搜索,我们可以找到超过 100 种类似工具,包括 Axivion、CAST、Coverity、Fortify、GramaTech、Klocwork、Lattix、Ounce、Parasoft、ProjectAnalyzer、ReSharper、SoArc、SofCheck、Viva64、Understand、Visual Studio Team System 以及 XTRAN 等。

单独来讲,每种静态分析工具支持最多 30 种语言。对于常见的编程语言,如 Java 和 C 语言,人们有数十种静态分析工具可以使用;对于古老一些的语言,如 Ada、Jovial 以及 PL/I 语言,人们只有很少一部分静态分析工具可以选择;而对于一些十分特殊的专用语言,如用于在 SAP 环境中编写代码的 ABAP 语言,人们只有一两种静态分析工具可以使用。

尽管没有进行任何详尽的调查,但在目前人们所开发出的约 2500 种编程语言中,静态分析工具大约可以支持其中的 50 种。然而,其中一些工具的规则具有可扩展性。因此在理论上,我们可以通过建立一些特定的规则来使得这些工具可以对所有的 2500 种语言进行检查。但是这几乎是不可能发生的,因为从经济学角度来说,对一些鲜为人知的语言或者不用于商业软件或科学软件的语言进行这种定制开发是不合算的。

整体来讲,静态分析工具是一种有效的方法,它可以发现软件中 85% 的常见编程错

误。因此，使用静态分析工具可以定义为一种最佳实践方法，同时其也正快速地成为一个通用的标准做法。

但是，静态分析工具只能发现代码中的问题，它并不能发现有毒需求、性能问题、用户界面问题以及某些种类的安全漏洞。因此，我们还需要引入其他形式的缺陷去除方法。

除了查找缺陷，一些静态分析工具还提供了其他一些功能。举例来说，一些工具可以帮助人们将代码从一种比较古老的语言转换成为一种比较现代的语言，如在需要时将 COBOL 语言的代码转换为 Java 语言。

我们还可以提高静态分析工具的应用范围，如用它来检查构成某些形式（如 UML）的需求文档或设计文档的元语言。事实上，使用某种形式对静态分析工具进行扩展后，我们可以建立一套软件测试的环境，这一点在理论上是可行的。

由于静态分析和正式代码审查所发现的缺陷通常是同样类型的，因此正常情况下人们会使用其中的一种，而不是全部使用。静态分析和审查在缺陷去除效率方面也几乎是同一水平，而静态分析所需费用更低，周期也相对短。但是代码审查能发现更多不易察觉的缺陷，如性能问题或者安全漏洞。这些问题本身并不算是“代码缺陷”，但是会引发一定的麻烦。

如果既使用了静态分析，也使用了代码审查（这种情况通常出现在一些重要的关键应用上，如医疗器械应用、某些种类的安全软件和军事应用等），那么我们通常会首先进行静态分析，再进行代码审查。

我们有时会发现静态分析工具所发现的一些问题事实上是“误报”，或者它认为某些代码段落是有缺陷的，结果我们发现这段代码是完全正确的。但是，考虑到静态分析有着如此之高的缺陷去除效率，这些“误报”可以认为是我们为此付出的一点小代价。

4. 子程序测试

测试有很多种，分别针对了不同的代码规模。其中的子程序测试是指针对一个非常小的代码集合（可能最多也就是 10 行指令）的测试，这个代码集合可能会产生一个需要验证的输出或者执行一个需要验证的动作。通常来讲，从代码规模角度，子程序测试是最底层的测试种类。

相比之下，单元测试通常会针对 100 行或更多指令来进行，而那些公共的测试种类如功能测试或者回归测试，则会处理上千行指令的代码集合。

随着源代码的规模增加和代码中路径的增加，为了能够对代码进行 100% 覆盖率的测试，我们所需要的测试用例也会越来越多。事实上，对于非常大型的软件系统，100% 的覆盖率通常是不太可能达到的，至少是非常罕见的。

子程序测试已经成为一种标准实践方法，同时也是一种最佳实践方法，这是因为它在去除软件问题方面有着非常显著的效果。然而，这种方法的实际缺陷移除率只有大约 30% 到 40%。这是因为这种方法所测试的代码段落非常小，人们很难通过这种方法去发现某些种类的缺陷，如代码分支错误。

子程序测试可能会使用正式的测试用例，也可能不使用。这种测试通常的执行模式是运行代码而后验证这段代码的输出。子程序测试的测试用例（如果有的话），通常是一次性的。

5. 手工单元测试

在所有通常意义下的个人测试形式中或者说所有由编程人员独立完成的测试中, 对一个完整的模块进行单元测试是最大规模的一种形式。和其他个人测试形式一样, 这种测试仅由程序员来完成而不会有其他人员如测试专家或者软件质量保证人员的参与。

手工单元测试是最早的正式测试方法, 也是最古老的测试方法。事实上, 在 20 世纪 60 年代和 70 年代早期, 那时的软件应用大约只包含 100 行左右的代码, 单元测试通常是唯一的测试手段。

单元测试阶段是指对于一个完整的模块所进行的测试, 一个模块通常大约会包含 100 行代码, 会执行一个独立的功能并包含独立的输入、输出、运算法则和逻辑, 而这些都是单元测试需要验证的。

单元测试可以混合使用两种测试形式: 黑盒测试和白盒测试。黑盒测试是指对于测试人员来说, 模块的代码是不可见的。测试人员只能看到输入和输出数据。因此黑盒测试主要是对输入和输出数据进行验证。而白盒测试是指模块的内部代码是可见的, 因此人们可以对软件代码中的分支和控制流程进行测试。混合使用这两种测试形式在理论上讲可以覆盖所有的测试点。但是, 通常情况下单元测试的代码覆盖率极少能达到百分之百, 甚至对于循环复杂度较高的大型软件项目来说, 其覆盖率会降到 50% 以下。

单元测试侧重于数据约束、数据范围、容错处理以及和安全相关的事项。不幸的是, 在寻找缺陷方面, 单元测试只有大约 30% ~ 50% 的效率。比如, 单元测试通常无法发现和性能相关的问题, 因为这类问题通常都在更长的代码路径中或者多个模块联合工作时才会出现。

对于那些包含了多个代码分支或者复杂流程的模块, 单元测试通常会在测试覆盖率上面出现一些问题。随着软件循环复杂度的增加, 为了测试能够覆盖所有路径, 我们会需要越来越多的测试用例。实际上, 对于循环复杂度超过 5 的软件模块, 即使这个模块仅仅包含 100 行代码, 人们也从未实现过百分之百的单元测试覆盖率。

单元测试已经成为软件工程项目中的一个标准活动, 因而尽管在某种程度上来讲它的缺陷去除效率相对较低, 单元测试仍然不失为一个最佳实践活动。如果没有单元测试, 那么后续的测试阶段, 如功能测试、压力测试、组件测试和系统测试都无从谈起。

单元测试所创建的测试用例通常会存储在一个正式的测试库中, 以便日后可以用来进行回归测试。由于这些测试用例倾向于长期重复使用, 因此这些测试用例需要一些明确的标识来记录这些测试用例所测试的应用类型、功能特性以及创建时间和创建人。同时, 我们还需要记录那些引用和执行了这些测试用例的相关测试脚本。正规测试用例设计的细节特性属于本书讨论范围之外的内容, 但读者可以在许多其他书籍中找到这类内容。

单元测试可以和其他形式的缺陷去除方法协同使用, 如正式代码审查或静态分析。通常情况下, 人们会在进行单元测试之前进行静态分析, 而在单元测试之后进行正式代码审查。这样做的原因是, 静态分析不仅执行速度较快, 费用相对较低, 而且能够发现很多在单元测试中可能会发现的缺陷; 而单元测试在代码审查之前进行也是出于同样的原因: 单元测试更快, 花费更低。但是, 代码审查能够发现一些逃过了静态分析和单元测试的隐藏

缺陷，如安全漏洞以及性能问题。

在同一段代码上同时使用代码审查、静态分析和单元测试这三种缺陷移除方法是非常罕见的。大多数情况下，这种情形只会出现在一些极端重要的软件应用类型上，如武器系统、医疗设备以及其他一些类型的应用上（通常这类软件如果失效，将会带来死亡或者毁灭性的后果）。

多年以来，手工单元测试一直是软件项目中一个常见的活动，即使在现在也仍然在广泛使用。但是，其效果却众说纷纭，有人说它效果“非常糟糕”，也有人用它效果“极好”。这主要是因为人们在执行单元测试的方法上和使用测试结果上的不一致，导致了其缺陷去除效率数据范围非常之大，以至于我们很难去判断这种方法在本质上到底是不是一个最佳实践方法。使用测试用例仔细进行黑盒测试和白盒测试并对测试覆盖范围进行仔细考量可以认为是一个最佳实践方法。而使用仓促完成的测试用例进行只覆盖了部分范围的单元测试只能称之为勉强应付的行为，而不能作为最佳实践方法。

测试是一种很有技术性的技能，因此很多学术单位和商业公司提供了很多测试培训课程。如果我们了解到正式的测试技术培训和认证是否可以明显地提高测试活动的缺陷去除效率，那么这种数据将会非常有用。有一些传说证实测试认证对于提高测试缺陷去除效率是有好处的，但是我们仍然需要对此进行更大范围的调查研究。

6. 自动化单元测试

尽管手工单元测试从20世纪60年代起就成为软件工程项目的一部分，自动化单元测试的出现却要晚得多。大约在20世纪80年代，随着软件应用越来越庞大和复杂，同时伴随着图形用户界面（Graphic User Interfaces, GUI）的出现，这极大地扩展了软件输入输出的类型，自动化单元测试开始出现在软件项目中。

在2009年前后，从某种程度上讲，“自动化单元测试”这个术语的含义并不明确。通常情况下，这个术语用来指代手工创建的单元测试测试用例以及一套可以按照一定规律自动运行并且不需要软件工程师介入的框架来执行这些测试用例。

敏捷开发社区和极限编程社区都开始使用自动化单元测试，这种应用伴随着一个结论，那就是在开始代码开发前进行测试用例的创建。这两种模式的结合由于成功地将软件工程师的关注点锁定在了质量领域，因而在缺陷去除方面有着非常显著的效率，同时在缺陷预防方面也取得了一定的成功。

自动化单元测试阶段主要是针对测试用例的执行以及所发现缺陷的记录和跟踪，大部分的测试用例仍然是手工创建的。但是，在理论上我们也可以设想通过自动化手段来创建测试用例的方式。

回想一下我们在第7章提到的，在需求收集和分析阶段，我们需要考虑7个基础性方面以及30个附加方面。巧合的是，这37个方面也是在测试中需要覆盖到的。任何一种用来对需求文档和功能说明书元语言进行静态分析的方法，在理论上都应该能够产生一个副产品，那就是一整套针对这些需求的测试用例。

一些自动化测试方法主要针对网络应用，另一些则主要针对嵌入式应用，还有一些会主要应用于信息科技产品。自动化测试作为一个新兴科技，在2009年的今天仍然处于快速

发展和壮大中。

我们可以在两个规模和复杂度相似的产品上分别执行自动化单元测试和手工单元测试来对这两种方法进行比较,但是截至目前,我们仍然缺少针对这种比较的可靠观察数据。有一些传闻为我们提供了一些自动化测试在执行速度和方便性上的优势。但是对于测试来讲,最主要的衡量指标应该是缺陷去除效率。在本书编写之时,我们在自动化测试方法和最佳手工测试方法的比较上并没有足够可靠的数据,因此也就不能去断定自动化单元测试比之手工单元测试是否有更高的缺陷去除效率。

随着越来越多关于自动化测试的信息和数据的出现,自动化单元测试方法有很大的机会可以成为最佳实践方法中的一员。截至2009年,数据显示自动化测试在人力投入和经费方面能够带来一些益处,但在缺陷去除效率方面带来的提高仍然无法确定。

7. 遗留应用的缺陷去除

软件工程行业中大约有40%的软件工程师都要面临这样一个问题,那就是如何对于那些并不是自己开发的古老遗留应用进行维护。尽管这些遗留应用历史悠久,但这并不意味着它们不会引起麻烦,相反,它们仍然有很多潜在的错误或缺陷。

这种情况为我们带来了很多问题,我们该如何对于那些原开发者已经无法找到的遗留代码进行缺陷去除。这些遗留代码的情况通常很复杂:功能说明书也许已经丢失或者即使还存在也已经过时;针对这些代码的注释也许很少甚至有些注释是错误的;这些代码本身是用一种已经死掉的语言所开发或者使用了一种维护团队所不使用的语言进行开发。

幸运的是,不少公司和工具已经注意到了这个维护古老遗留应用方面的问题。其中的一部分公司开发出了一些“维护工作台”,主要包含以下功能:

1. 自动化静态分析
2. 自动化测试覆盖率分析
3. 自动化功能点数量计算
4. 自动化循环复杂度和基本复杂度计算
5. 自动化调试(可以针对许多语言进行,但不是所有编程语言)
6. 遗留应用中商业规则的自动化数据挖掘
7. 从已死掉的语言向较新的语言的自动化翻译

考虑到这些古老的遗留应用是用多达2500种不同的编程语言所开发的,没有任何一种工具可以对这些遗留应用提供一种通用的解决方案。但是,对于那些使用了较为常用的语言,如Ada、COBOL、C、PL/I等类似语言所编写的遗留代码,我们有很多可用的工具可以进行处理。

我们可以将使用维护工作台这类工具作为一个最佳实践方法,考虑到工具的多样性以及各个工具间差异很大,我们很难将使用某种特定的工作台作为最佳实践。同时,这些工具仍然在快速发展当中,并且经常会有新功能出现。

8. 个人缺陷去除方法的联合应用与协同配合

这部分所讨论的缺陷去除方法通常都不是单独使用的,人们更加倾向于协同使用其中的若干种。最常见的协同使用形式是联合使用调试、自动化静态分析及单元测试。有经验

的软件工程师通过联合使用这三种方法，可以将缺陷去除效率提高至 97%；但是对于新手来讲，联合使用这三种方法也可能仅有 85% 的缺陷去除效率。

9. 个人缺陷去除方法小结

尽管由于个人缺陷去除方法的相关活动都是由个人来进行的，因而我们很难去进行研究，但是 50 多年以来，个人缺陷去除方法始终是同软件缺陷战斗的前沿阵地。这就是说，软件产品会出现缺陷并且在软件投放市场后仍然存在缺陷，这一现象说明没有任何一个个人缺陷去除方法可以达到 100% 的去除效率。

然而，一些新的缺陷去除工具如自动化静态分析工具正在改变这种状况，并使个人缺陷去除方法和工具系列变得更加严格和正规。

鉴于每个软件工程师会对他们自己在个人缺陷去除活动中发现的缺陷进行记录，通过这些记录数据，如果我们能够在公开缺陷去除活动开始前，将个人缺陷去除方法的效率提高到 90% 以上，这将会是一个非常有价值和作用的成就。

在软件工程学从一门手艺成长为一个真正工程学科的过程中，个人缺陷去除方法仍将会担任一个十分重要的角色。了解哪种方法在缺陷预防和缺陷去除方面是最有效率的和最有效果的，是软件工程职业化水准的一个标志。如果我们对于缺陷去除方法的效率缺乏测量标准，或者对于各种方法的缺陷去除效率没有明确的了解，那只能说明软件工程仍然处于业余水平，还不具有职业水准。

8.14 “代码行”度量方法的经济学问题

1. 简介

任何针对软件编程和代码开发的讨论中，如果没有涉及著名的代码行（Lines of Code, LOC）度量方法，都不能算是完整的。人们从计算机时代的开始之初便使用这种度量方法测量软件项目生产率和软件质量。

代码行度量方法最初出现是在 1960 年前后，其主要用来研究软件的经济效益、生产率和质量的。最初，代码行度量方法在以上三个方面确实是相当有效的。

但是随着越来越多的高级语言的出现，代码行度量方式也开始出现问题。代码行度量方法并不能对那些非编码活动进行测量，如需求阶段和设计阶段的活动，而这些活动所需的费用正变得越来越高昂。

代码行度量方法中所包含的问题变得越来越严重。在一项 1994 年进行的对照研究中，研究人员对使用 10 种不同的编程语言所编写的相同软件应用，分别使用代码行和功能点数量两种度量方式进行测量，研究结果令人十分震惊：代码行度量方式所得出的结果严重偏离了软件项目经济生产率的标准预设结果，对于包含超过一种编程语言的研究来讲，这种偏离的严重程度足以将使用代码行方式进行度量视作玩忽职守和渎职。

如果没有足够的例证和实例研究来证明代码行度量方式的问题，人们是不会得出如此严重的结论的。以下我们将通过按照时间顺序来介绍代码行度量方式的使用历史，展示出这种度量方法从何时开始变得不再有用并开始引发问题，以及为何会如此。这部编年史从

20 世纪 60 年代一直到今天，并叙述了我们对 2020 年左右的软件世界的一些见解。

2. 代码行度量方式在 1960 年左右的应用

代码行度量方法最初出现是在 1960 年前后，其主要是用来研究软件的经济效益、生产率和质量的。软件应用的经济效益使用“每行代码的费用”来衡量，而生产率则使用“单位时间内的代码行数”进行测量，软件的质量使用“每千行代码中的缺陷数量”(Defects Per KLOC, K 用来指代 1000 行代码)来衡量。代码行度量方法在以上三个方面确实是相当有效的。

当代码行度量方式刚刚出现的时候，人们只有一种编程语言可以使用，那就是基础汇编语言。软件程序通常都很小，而编程工作占据了整个软件项目工作的 90%。对于基础汇编语言来说，物理代码行和逻辑代码行所代表的含义是一样的。

在这种早期软件行业环境中，代码行度量方式对于分析软件的经济效益、生产率和质量都是相当有效的。如果人们只有一种编程语言，程序所使用的可重用代码极少甚至根本没有，而且物理代码行数量和逻辑代码行数量并没有明显的区别，那么人们使用代码行度量方式将会非常得心应手。在这个代码行度量方式的黄金时期，这种测量方法非常有效并且没有任何竞争对手。但是，这个黄金时期仅仅持续了大约 10 年。

然而，10 年的跨度并不短暂，这段时间足以将代码行度量方式深深植入软件工程师的意识之中。一旦这种意识扎根之后，除非新的证据成为压倒一切的历史趋势，否则我们很难将其从思想中除去。遗憾的是，尽管软件工程行业在不断地变化发展之中，代码行度量方式却并没有变化。随着时间的流逝，代码行度量方式的价值越来越低。甚至在 1980 年左右时，这种度量方式已经产生了相当严重的负面影响，但是并没有太多的人意识到这一点。由于认知上的不一致，人们在使用代码行度量方式时并没有随着其他软件工程学方法的改变而对这种度量方式进行仔细检查，而是草率地继续使用它。

3. 代码行度量方式在 1970 年左右的应用

到了 1970 年，宏汇编语言已经取代了基础汇编语言。同时，第一代高级语言，如 COBOL、FORTRAN、PL/I 语言也已经开始应用于软件行业。由于这些更好的替代品的出现，基础汇编语言的使用率开始下降。这应该是软件工程历史上第一个一系列编程语言开始淡出人们视线的例子，随着这些语言的淡出，留给人们的是众多难以维护的遗留应用，随着能够使用这些无人问津的编程语言的程序员以及相应的编译器逐渐减少直至消失，对这些应用的维护也就越来越困难。

1970 年，目前已知的代码行度量方式的第一个问题出现了。IBM 的很多出版物组织的支出超出了当年的预算。笔者发现，通常科技出版物组织的预算是基于软件编程或编码预算的 10% 来进行分配的。

这种出版物项目预算和编码预算相联系的方式在使用汇编语言时并不会发生超支，但是对于 PL/S 语言 (PL/I 语言的一个变体) 项目手册的出版工作来说，通常出版费用会有很大的超支现象。这是因为 PL/S 语言可以将编码的工作量减半，但是技术手册编写的工作量仍然和原来一样。因此，如果出版预算设置为编码预算的十分之一，而编码费用可以节省 50%，那么 PL/S 语言项目的所有出版费用就必然要发生超支。

针对这个问题，IBM 最初的解决方案是为每种编程语言的等级赋予一个正式的数字定义。编程语言等级数字的含义是，针对同样的功能，等价于此种高级语言一行代码的基本汇编语言的代码行数。由此定义，COBOL 的等级是 3，因为 3 行基本汇编语言代码等价于 1 行 COBOL 代码。使用同样的规则，SMALLTALK 语言的等级是 18。

在人们发明功能点计算方法之前的多年时间里，IBM 一直在使用这种“等效汇编语句”方法作为估算非编码工作（如用户手册编写）的基本方法（事实上，即使在 2009 年，仍然还有一些公司在使用等效汇编语言方法）。

因此，对于一个使用 PL/S 语言编写的软件来说，人们不再使用代码编写工作量的 10% 作为出版物编写工作量的估算基础，而是将代码编写工作量转化为使用基础汇编语言的工作量，再使用它的 10% 作为出版物编写的工作量估算基础。这种方法虽然听起来很原始和简单，但是在实际使用中却相当有效果。这种方法意识到了这样一个问题，那就是对于同样的一个功能来说，不同的语言所需要的代码行数量是不一样的。

但是，无论是 IBM 的客户还是 IBM 的高层管理人员对于这种将现代高级语言的规模等效转换为一种已经过时语言的规模以便进行费用估算的做法感到无法接受。因此，人们有必要开发一种新的测量方式。

出版物工作量方面的问题，以及对于等效汇编语言方法的不满，是 IBM 决定派 Allan Albrecht 和他的同事来开发功能点度量方法的两个主要原因。另外，随着一些功能非常强大的编程语言（如 APL 语言）的出现，IBM 需要一种度量方法和估算方法可以对编码工作和非编码工作的规模和费用进行准确的评估。

宏汇编语言的出现和使用也为人们引入了代码重用的概念，同样，这也引起了规模测量方面的问题。这样的问题主要包括两个方面：一个是如何计算软件中重用代码的规模，另一个就是如何计算其他重用资料的经济效益。

上述问题的解决方案，就是将软件的生产率划分为两个不相关联的方面：

1. 软件开发生产率
2. 软件交付生产率

前者，软件开发生产率，主要讨论的是代码以及其他一切需要使用传统方式从草稿逐渐完善的材料。

后者，软件交付生产率，主要讨论的是最终交付的软件应用，包括所有重用的材料。举例来说，对于一个使用了宏汇编语言的程序来说，软件开发生产率的基数可能是每个月 300 行代码。但是考虑到人们通过宏扩展的方式使用了一些重用代码，其产品交付生产率可能高达每个月 750 行代码。

即使是在 2009 年，人们仍然没有广泛地意识到软件行业的这种重要商业特质。软件工程行业的真正目标，应该是提高软件项目的交付生产率。事实上，在软件交付生产率提高的同时，软件开发生产率有可能会下降。

为什么会发生这种情况呢？这种情况的发生，可能是由于人们对一个重用代码组件的精心开发以及对其进行零缺陷等级的认证。假设我们有一个 500 行代码的重用组件需要开发，而开发完成后这个组件可以得到非常广泛的使用。我们非常仔细地开发这个组件并进

行全面的审查、静态分析检查以及全面的测试，以便将这个软件达到零缺陷等级的认证。

这种精细开发和全面认证可能会将软件的净开发生产率下降到仅仅每个月 100 行代码，而通常情况下一个一次性模块的开发生产率接近每个月 500 行代码。因此，我们需要总共五个月的时间，而不是仅仅一个月，来开发这样一个 500 行代码的模块。这当然是一个非常低的软件开发生产率。

但是，一旦这个模块经过了认证并作为可重用组件提供给其他软件。假设在软件中使用这个模块的工作可以在仅仅一小时内完成，那么，每次人们使用这个模块，它便为人们节省了大约一个月的开发时间。

如果这个模块仅仅被重用了 5 次，它便已经补偿了在开发阶段的低开发生产率所带来的损失。每次人们使用这个模块时，它的有效交付生产率等同于每小时 500 行代码，或者说大约每个月 66 000 行代码。

因此，在这个模块的软件开发生产率降低到了仅仅每个月 100 行代码的同时，其等效的软件交付生产率却达到了每个月 66 000 行代码。由此可见，这个模块的真正经济价值并不在于其开发速度有多快，开发周期有多短，而是在于这个可重用模块在其他软件应用中的重用次数。

为了确保这种重用的成功，这些重用代码需要接近或者达到零缺陷的状态。这些模块自身的开发速度并不重要，因为其开发一旦完成，这些代码便可以应用到数百个软件应用中。

对于面向服务的架构（SOA）和软件即服务（SaaS）方案来说，它们的目标就是在软件交付能力上取得显著提高。只要能够保证软件的质量能够达到零缺陷的水平，软件开发速度相对来说不那么重要了。

回到软件业的历史编年表中来，宏汇编语言和其他新编程语言所导致的另一个问题就是物理代码行数量和逻辑代码行数量之间的区别。一些语言，如 Basic 语言，允许在同一行物理代码中出现多个指令。还有一些语言，如 COBOL 语言，将一些逻辑代码行分割为多行物理代码。物理代码行数量和逻辑代码行数量之间的差值最高可达 500%。对于一些语言来说，物理代码行的数量可能比逻辑代码行数量要多，而对于其他语言来说，情况正好相反。代码行度量方式的使用者们从来都没有彻底解决这个问题。直到 2009 年的今天，这个问题仍然会引起一些麻烦。

随着高级编程语言，如 C++、Objective C、SMALLTALK 等，变得越来越强大，同时也越来越复杂精细，软件项目中编码工作所占的比例从 90% 下降到了大约 50%。由于编码工作所占比重的下降，代码行度量方式对于研究软件的经济效益、生产率和质量已经不再有效了。

人们在 1975 年左右开发出了功能点度量方法之后，对于编程语言等级的定义中加入了此种语言等效于 1 个功能点的逻辑代码指令的数量。举例来说，当我们使用 COBOL 语言时，对于过程和数据部分的单个功能点的开发需要大约 105 条指令。

这种对于编程语言等级定义的扩展构成了逆火系统的数学基础，或者说从源代码直接转化为功能点数量的计算基础。当然，各不相同的编程风格使得逆火系统的计算结果很不准确，但尽管如此，对于那些有代码却没有使用说明书的遗留应用来说，人们仍然广泛使

用逆火系统来进行计算。

一些咨询公司,如 David Consulting、高德纳咨询公司(Gartner Group)和软件生产力研究所(Software Productivity Research, SPR)都还有一些空缺的职位在进行招聘。这些职位主要针对数百种编程语言进行研究,以发现使用每种语言编写一个功能点所需代码指令的数量。

1978年在加利福尼亚州蒙特雷市举行的一场由IBM、SHARE和GUIDE联合主办的会议上,A.J.Albrecht在他的一个公开演讲中为大家介绍了功能点度量方法。在此之后,功能点度量方法很快便出现在各大软件文献中。很快,IBM的客户开始使用这一方法,而这也导致了加州一个功能点用户组的建立。

4. 代码行度量方式在1980年左右的应用

大约1980年的时候,编程语言的数量已经达到了50种。同时,面向对象语言也在快速发展中。因此,软件代码的重用现象也在快速增加。

在1980年左右出现的另外一个问题是,许多软件应用开始使用超过一种编程语言进行编码,如在一个软件产品中同时使用COBOL语言和SQL语言。这种在同一个软件应用中使用多种编程语言的趋势很快变成了行业标准,而不仅仅是一些个例。然而,一个软件中使用多种语言也为准确计算代码行的数量增加了一定的困难。

在20世纪80年代的中期,功能点度量方法的用户们组织和建立了一个非营利组织,国际功能点用户组(IFPUG)。这个组织创始于加拿大,而后在20世纪80年代中期迁至美国。随后相继在其他国家建立了分支机构,截至20世纪80年代末时,功能点用户组已经遍布世界上十几个国家。

1985年,第一个对软件费用进行估算的商业软件开始投入市场。这个软件的名字叫做SPQR/20。它支持对30种常见的编程语言进行估算,同时也支持那些混合使用了多种编程语言的软件。

该工具可以对软件项目的文档,如需求文档、设计文档和用户手册,进行规模估算和费用估算。同时,它也可以对一些非编码工作,如测试工作、项目管理工作,进行估算。

考虑到人们仍在广泛使用代码行方法,SPQR/20工具在表述软件项目的生产率和质量时,同时使用了功能点方法和代码行方法。使用这个工具时,我们可以很容易地将估算对象从一种语言切换到另外一种语言,因此它可以帮助我们得到并比较不同语言,如宏汇编语言、FORTRAN、Ada、PL/I或者Java语言的估算结果,这些结果都使用功能点方法和代码行方法同时进行描述。而对比不同语言的结果也可以帮助我们得到一些有趣的结论。

随着编程语言等级的提升,使用每个人月所完成的功能点数量来描述的项目经济生产率也会相应提升。这一点是符合标准的经济学理论的。但是随着语言等级的提升,用每个月所完成的代码行数量来描述的经济生产率却在下降。代码行度量方式所表现出的这种反常现象违背了所有的标准经济学理论,这也是我们断言使用代码行度量方式几乎算是渎职的主要原因。

在制造业经济学中,有一个广为人知的规则,即当一个开发周期的费用中有较高比例的固定费用时,如果所制造元件的数量减少,那么每个元件所需要的制造费用将会上升。

如果我们将代码行考虑为一个制造元件,当我们从较低等级的语言转换为使用高级语言的时候,所制造元件的数量会减少。但是各种形式的文档工作(包括需求文档、使用说明书以及用户手册等)并没有减少,这部分的费用几乎是作为一个常量不会发生变化,因此在经济学上其等同于固定费用。理所当然,这会使每个元件所需的费用有所上升。这种描述也许难以理解。下面我们通过两个例子来进行讲解。

例 A 假设我们有一个软件应用由 1000 行基础汇编语言代码组成(我们可以假设这个软件应用有 5 个功能点)。同时假设开发人员的薪酬是每个人月 5000 美元。

假设编码工作需要 1 个人月来完成而文档编写工作(包括需求文档、产品说明书以及用户手册)同样也需要 1 个人月。项目整体需要 2 个人月来完成,其所需费用为 10 000 美元。如果我们使用每个人月所完成的代码行数来描述生产率,那么此时的数值为 500。每行代码所需费用为 10 美元。使用每个人月所完成的功能点数量来描述的生产率数值为 2.5。每个功能点所需费用为 2000 美元。

例 B 假设我们使用 Java 语言来开发和例 A 中相同的一个软件应用。不同于例 A 中的 1000 行基础汇编语言代码,Java 版本的程序仅需要 200 行代码。软件的功能点数量仍然是相同的 5 个而开发人员的薪酬也仍然是每个人月 5000 美元。

在这个例子中我们假设编码工作仅仅需要一个编程人员一周的工作,即约为 0.25 个人月。但是文档编写工作仍然需要 1 个人月的工作量。

现在我们来,整个项目仅需要 1.25 个人月,而不是 2 个。整体费用也不再是 10 000 美元而仅有 6250 美元。很明显,软件的经济生产率提高了。因为我们完成了和例 A 中同样的工作但是却节省了 3750 美元的费用。和例 A 相比,我们提供给用户完全同样的功能,但是所编写的代码更少,因此所需的人力就更少,因此我们真实的经济生产率是有提高的。

但是如果我们使用代码行度量方式来考察整个项目的生产率,我们发现项目生产率从例 A 中的每个月完成 500 行代码降低到了每个月仅仅完成 160 行代码 ($200/1.25$)。

再来看项目的单位代码行费用,这一数值升高到了每行代码花费 31.25 美元。很明显,代码行度量方式并不能显示出软件项目真实的经济生产率。另一个明显的事实是,代码行度量方式使得高级语言在生产率比较中处于不利地位。事实上,很多研究已经证明,代码行度量方式对于测量结果的偏离程度和编程语言的等级是成比例的,也就是说,编程语言等级越高,其代码行度量方式的结果就越糟糕。

我们再看功能点度量方式,由于例 A 和例 B 中的软件都包含 5 个功能点,例 B 的生产率是每个人月完成 4 个功能点。每个功能点所需要的费用仅仅是 1250 美元。这些数据所显示出来的生产率提高和标准经济学理论的规则是一致的,因为对于同一个软件来讲,开发速度快、费用低的方式理应比开发速度慢但费用却更加高昂的方式拥有更好的生产率数据。

在上述例子中有一个现象,那就是即使代码编写部分的工作量已经显著减少,文档编写部分的工作量也并没有减少。这就是为什么代码行度量方式用来对不同编程语言的软件项目进行比较时会产生违反行业规则的结果。它完全和标准经济学中的生产率规则背道而驰,因而使得高级语言在比较中处于不利地位。表 8-7 为我们总结了例 A 和例 B 的各项结果并进行对比。

表 8-7 初级语言和高级语言的比较

	例 A	例 B	差值
编程语言	汇编语言	Java 语言	
代码行数	1 000	200	-800
功能点	5	5	0
员工月薪	5000 美元	5000 美元	0 美元
文档编写工作量(人月)	1	1	0
代码编写工作量(人员)	1	0.25	-0.75
总工作量(人月)	2	1.25	-0.75
项目总费用	10 000 美元	6250 美元	-3750 美元
单位月份所完成的代码行数	500	160	-340
每行代码的费用	10 美元	31.25 美元	21.25 美元
单位月份所完成的功能点数	2.5	4	1.5
每个功能点的费用	2000 美元	1250 美元	-750 美元

通过对例 A 和例 B 进行对比我们会看到,代码行度量方式实际上将经济学方程式完全颠倒,使得笨重、缓慢、昂贵的方式看起来比较便、快速、经济的方式更好。

可能有人会说,代码行度量方式对于生产率比较结果的颠倒是因为我们将代码编写工作和文档编写工作一起进行统计和计算。但是即使我们仅仅计算代码编写工作自身的数据,代码行度量方式仍然会违反标准经济学的假设。

软件工程师们可以在一个月完成 1000 行汇编代码的编写,因此其生产率是每月 1000 行代码。编码工作的费用是 5000 美元,或者说每行代码 5 美元。

同样,软件工程师们可以用一周的时间完成 200 行 Java 代码,也就是 0.25 个月。我们将其转化为月度的数据,也就是每个月仅仅 800 行代码。Java 语言版本的开发费用是 1250 美元,因此其每行代码的费用是 6.25 美元。

因此,对于单位代码行的费用来说,Java 语言要高于汇编语言,尽管 Java 语言版本的开发仅仅用了汇编语言版本四分之一的的时间,并且总编码费用也仅仅是其四分之一。当你试图使用代码行方式来自比较两种不同的语言时,你会发现汇编语言要优于 Java 语言。这当然是一个错误的结论。表 8-8 展示了当仅统计编码工作时,汇编语言和 Java 语言的对比结果。

表 8-8 初级语言和高级语言的比较(编码部分)

	例 A	例 B	差值
编程语言	汇编语言	Java 语言	
代码行数	1000	200	-800
功能点	5	5	0
员工月薪	5000 美元	5000 美元	0 美元
代码编写工作量(人员)	1	0.25	-0.75
代码编写总费用	5000 美元	1250 美元	-3750 美元

(续)

	例 A	例 B	差值
编程语言	汇编语言	Java 语言	
单位月份所完成的代码行数	1000	800	-200
每行代码的费用	5 美元	6.25 美元	1.25 美元
单位月份所完成的功能点数	5	20	15
每个功能点的费用	1000 美元	250 美元	-750 美元

从真实的经济学角度来看, Java 代码编写费用是 1250 美元而汇编语言代码编写花费了 5000 美元。很明显, Java 语言有着更好的经济效益, 因为其在完成同样工作的同时节省了 3750 美元。

但是对于代码行生产率来讲, Java 语言的数据要低于汇编语言, 而且每行代码的费用由汇编语言的 5 美元增加到了 Java 的 6.25 美元。从经济学角度来看, 如果同一个项目在使用不同语言时所需要的代码数量有较大差异的话, 在对这两种语言的生产率进行比较时, “单位月份所完成的代码行数”和“每行代码的费用”这两个指标的差异并不重要。

遗憾的是, 当我们试图使用代码行度量方式来衡量两种不同编程语言的经济生产率时, 无论我们用何种方式来进行计算^①, 代码行度量方式最终都会引起测量结果的颠倒。相比之下, 当我们计算功能点时, Java 语言每个功能点的开发费用是 250 美元而汇编语言的单位功能点开发费用是 1000 美元, 这一结论符合标准经济学的假设。

Java 语言的功能点生产率是每个人月 20 个功能点, 而汇编语言每个人月仅仅对应 5 个功能点。因此, 功能点度量方式的测量结果更加符合标准经济学的假设, 而代码行度量方式违背了标准经济学。

回到我们最初的讨论主线上来, 在 SPQR/20 工具之后的几年内, 其他所有软件规模估算的商业工具都开始支持功能点度量方式, 因此 CHECKPOINT、COCOMO、KnowledgePlan、Price-S、SEER、SLIM SPQR/20 以及其他一些工具都会同时使用功能点方式和代码行方式来描述其估算结果。

到了 20 世纪 80 年代末的时候, 编码工作的工作量在总项目工作量中所占的比例已经低于 35%, 而此时代码行度量方式无论是对于经济效益研究还是质量研究都已经不再适用。代码行度量方式无法确定需求缺陷和设计缺陷的数量, 而这两种缺陷在当时已经多于编码缺陷。代码行度量方式不能用来测量任何非编码活动, 如需求、设计、文档编写和项目管理活动。

而代码行度量方式的使用者们对于以上这些问题所采取的措施也令人叹息: 他们仅仅是不再使用代码行方式去测量除了代码编写和代码缺陷之外的工作。在所有公开发布的使用代码行方式进行测量的项目报告中, 大部分都仅仅覆盖了不到 35% 的软件开发工作和不到 25% 的软件缺陷, 同时几乎没有任何关于需求缺陷和设计缺陷的公开数据, 而需求蔓延率、设计费用和其他一些新出现问题方面的数据也极少。

① 如之前的统计代码编写工作和文档编写工作, 以及仅统计代码编写工作自身。——译者注

代码行度量方式的历史为 Leon Festinger 博士的认知失调理论提供了一个非常有趣的例证。这个理论说,一种认知一旦确立,人们便会倾向于拒绝所有与之相悖的例证来否定新的认知。只有当新认知的例证越来越多以至于已经不可阻挡的时候,人们才会接受它并改变自己的观点。这种认知的改变一旦发生,便会在很短时间内完成。

5. 代码行度量方式在 1990 年左右的应用

大约 1990 年的时候,编程语言的数量已经达到了 500 余种。不仅如此,部分软件应用的开发过程使用了 12 到 15 种不同的编程语言。但业内并没有一个就如何计算代码行的数量产生一个国际标准,各种计算方法也使用了不同的变量,其中一些甚至并没有经过严格的定义。

1991 年,在笔者的著作《Applied Software Measurement》的第一版中,作者初步提出了一个计算代码行的方案,那就是使用逻辑代码的数量来计算代码行数量。一年之后,来自软件工程研究所(Software Engineering Institute, SEI)的 Bob Park 也发表了一个初步的计算方案,其计算思路是仅仅根据代码的物理行数来计算。

笔者在 1993 年对软件期刊中的文章进行过一次调查,结果显示其中三分之一的文章使用物理代码行数来计算,另三分之一的文章使用逻辑代码行数,剩下的三分之一在使用代码行度量方式的时候甚至没有提到如何去计算代码行数。考虑到对于很多语言来说,逻辑代码行数量和物理代码行数量的差异可能高达 500%,这种情况并不是一个好现象。

在医学工程和医学实验类的技术期刊中,人们通常会花费高达 50% 的篇幅来叙述和定义实验人员是如何对实验结果进行测量的。但另一方面,软件工程学期刊却完全没有定义其数据结果的测量方式。

通常软件工程学期刊的文章仅仅用数行文字来描述实验结果测量方法的细节,极少有文章会用更多的篇幅来描述。这也是使得人们认为“软件工程”这个术语本身就是自相矛盾的原因之一。事实上,在某些国家,使用“软件工程”这个词甚至是非法的,因为软件开发并不是一个公认的工程学科或是一个得到许可的工程学科。

但是,除了计算代码行数量方法的不明确之外,我们还有一个更加严重的问题。Visual Basic 语言的诞生为我们带来了一系列新的语言,对于这些语言来说,计算其代码行数几乎是是不可能的。因为很多 Visual Basic 的编程工作并不是通过编写过程代码来完成,而是使用按钮和下拉菜单来完成的。

截至 2009 年,世界上大约有 2500 种编程语言及其变体语言,在这 2500 种语言中,约有 150 种是有公开有效的代码行计算规则的。还有大约 2000 种和其他语言有多种程度的相似,因此可以使用同样的计算规则。但是至少有 50 种语言是使用图形或其他可视化的菜单来产生代码,以此作为程序化代码的补充。对于这些语言来说,人们根本没有任何规则可以对其代码数量进行计算。而更加不幸的是,这些无法计算代码行数量的语言中的一部分往往是网站开发项目中最常使用的新式编程语言。

1994 年,人们对于代码行度量方式和功能点度量方式进行了一次对照研究。这项研究使用了 10 种不同的编程语言来开发同样的一个软件应用。这 10 种语言中,包括了 4 种面向对象语言。

研究结果在 1994 年的《American Programmer》上发表。研究发现,代码行度量方式背离了经济生产率中的基本概念,由于高级语言和面向对象语言的项目中存在固定的需求、设计及其他非代码工作的费用,因而代码行测量方式将这些语言在经济效益比较中置于不利的地位。这是第一个证明了代码行测量方式中所包含的问题的公开研究,它指出如果一个软件开发项目中使用了超过一种编程语言,那么使用代码行度量方式来研究其经济价值实际上是不负责任的。

到了 20 世纪 90 年代,大多数收集软件项目标准和基准数据的咨询业研究项目都使用功能点数据。没有任何大型的标准再使用代码行数据。1997 年,International Software Benchmarking Standards Group (ISBSG) 组织建立,其发布的内容仅使用功能点数据。咨询公司,如软件生产力研究所 (SPR) 和 David Consulting Group,同样也使用功能点数据。

20 世纪 90 年代末期的时候,有一些项目,其编码工作所占的工作量不到项目总工作量的 20%,因此代码行度量方式无法用于对那部分占 80% 工作量的非代码工作进行考量。但代码行度量方式的使用者们却仍然无动于衷,对于代码行方式的问题视而不见并坚持使用它来考量编码工作,而完全不去考量整个开发过程(包括需求收集、分析、设计、用户文档编写、项目管理工作以及其他种种非编码工作)的经济效益。

同时,在 20 世纪 90 年代末期的时候,在需求和设计阶段的非编码缺陷数量已经超过了编码阶段的缺陷数量,其比例几乎达到了 2:1。但是由于代码行度量方式并不能考量这些非编码缺陷,那些代码行度量方式的文献选择忽略掉这些缺陷。

事实上,即使是在 2009 年,人们仍然在就代码行方式的用处进行争论,但是不幸的是,所有的争论并没有坚持围绕制造业的经济规则来进行。看起来代码行方式的支持者们似乎完全忽略掉了软件开发过程中那部分固定费用的影响。

代码行方式的支持者们的一个观点是,软件产品开发的工作量和使用代码行方式计算的软件规模有着严格的统计关联性。确实如此,但从标准的经济效益角度看,它们又是不相关的。

如果一个包含 10 个功能点的软件需要 1000 行 C 语言代码,其开发费用是 10 000 美元,那么每行代码的费用是 10 美元。假设开发这个软件需要一个月的工作量,那么使用代码行方式计算的生产率是每月完成 1000 行代码。

如果同样这个 10 个功能点的软件使用 Objective C 来开发的话,可能我们只需要 250 行代码,而开发费用仅需要 2500 美元。开发的工作量可能仅需要一周,而不是一整月的时间。但是如果我们使用代码行来衡量的话,其每行代码的费用并没有变,仍然是 10 美元,而且代码行方式的生产率也保持不变,仍为每月完成 1000 行代码。

从使用代码行度量方式来看,以上两个版本的开发方式有着同样的生产率(每月完成 1000 行代码),但是这仅仅是软件开发生产率而不是软件交付生产率。考虑到使用 C 语言和使用 Objective C 的方式都是包含了 10 个功能点的开发,我们必须要注意到,C 语言开发方式的单位功能点费用是 1000 美元,而 Objective C 语言开发方式的单位功能点费用仅为 250 美元。

如果从每月完成的功能点来考量,C 语言版本的数据是 10 个,而 Objective C 语言版本

的数据提高到了 40 个。因此, 如果使用正确的度量方式, 高级语言在经济价值和交付生产率方面的优势便可以体现出来, 而代码行度量方式完全无法告诉我们这两者在经济效益或交付生产率方面的区别。

6. 代码行度量方式在 2000 年左右的应用

到 20 世纪末的时候, 编程语言的数量已经超过了 2000 种并以每个月超过一种新语言的数量增加着。现在, 新编程语言的出现速度可能已经达到了每年 100 种。

网络应用在迅速增加, 而所有这类软件应用都使用高级语言所开发, 并十分依赖充足的 reusable 组件。敏捷开发方式也在迅速发展并且同样倾向于使用高级语言。在某些软件应用中, reusable 组件部分的比例达到了 80%。代码行度量方式无法应用于大多数网络应用, 当然也不适用于考量敏捷开发方式中的 Scrum 会议和其他非编码活动。

功能点方式已经成为正式的软件项目经济效益研究和质量研究的主流度量方式。但是功能点方式仍然存在两个方面的问题, 正是这两个问题使得功能点度量方式无法正式成为经济效益研究和质量研究的行业标准方式。

第一个问题源于业内越来越多的大型软件项目。由于一些软件的规模甚至超过了 30 万个功能点, 如果使用通常的功能点分析方法, 不仅速度慢, 而且其费用也会过于高昂。

事实上通常的功能点分析方法对于软件规模的两极都有短板。如果一个软件的规模超过了 15 000 个功能点, 那么计算其功能点个数的活动不仅周期过长, 而且费用也过于高昂, 因此对这类项目通常从来没有正式计算过功能点的准确数量 (功能点分析的每个进程每天可以统计 400 ~ 600 个功能点, 而每计算一个功能点所需要的费用大约是 6 美元)。

而在规模的另一端, 功能点计算方法的规则并不适用于低于 15 个功能点规模的项目。因此, 软件的小幅度修改或者缺陷修复的规模无法计算。如果我们去看其中一个项目, 这类修改可能会小到只有五十分之一一个功能点, 即使规模较大的也极少超过 10 个功能点。但对于大型公司来说, 它们每年可能会有 30 000 个甚至更多的类似修改, 其总规模可能会超过 10 万个功能点。

第二个问题就是, 由于最初的功能点度量方式的成功, 引发了人们在其最初版本的基础上竞相开发不同的功能点度量方式。截至 2009 年, 我们一共有至少 24 种不同的功能点度量方式。这使得使用功能点方式进行标准研究和基准研究变得很困难, 因为我们几乎没有转换工具可以帮助我们一种功能点度量方式转换为另外一种。

除了最初的 IFPUG 标准功能点方法外, 人们还开发出了 Mark II 功能点方法、COSMIC 功能点方法、荷兰功能点、荷兰功能点、故事点、特性点、Web 对象点等许多种方法。

尽管人们仍在使用代码行度量方式, 这种方式仍然存在很严重的问题, 即如果一个软件开发项目中使用了超过一种编程语言, 或者项目的非编码工作中存在显著问题, 那么使用代码行度量方式来研究其经济价值或者软件质量实际上是不负责任的。

对于代码行方式的使用者来说, 同样还有心理上的问题需要改变。这些使用者更倾向于将注意力集中在编码工作而对其他软件项目工作视而不见。对于大型软件项目来说, 从事非编程工作的人员可能会比编程人员多很多。这些人员可能是软件架构师、设计师、数据库管理员、质量保证人员、技术文档编写人员、项目经理以及其他各工种。但是因为代

码行方式无法对以上任何一种工作进行考量，代码行方式的文献选择忽略这些工作。

7. 代码行度量方式在 2010 年左右的应用

如果我们能够对未来做出一个乐观的预测，那当然是皆大欢喜的。但是经济危机的出现改变了行业的现状并使得未来变得不确定起来。

如果维持目前的发展趋势，在未来几年之内，在软件工业中将会存在超过 3000 种编程语言，而其中大约 2900 种语言都已经被废弃或即将成为无人使用的语言。在软件工业中将会存在超过 20 种计算软件代码行数量的方法，超过 50 种计算功能点数量的方法，可能还会有超过 20 种其他不可靠的度量方法，如故事点、用例点、平均缺陷成本或者使用某种未知数据的百分比方式（软件行业经常会发布一些类似“将生产率提高了 10 倍”的声明，却并未定义所比较的数据内容）。

毫无疑问，未来的社会学家会对软件工程行业的现状十分感兴趣。他们会奇怪为什么软件行业花费了如此多的精力去建立各种不同形式的方法，但却不愿意花费力气去制订一些基础性的标准。毫无疑问，在未来，大型的项目仍然会在进行到一半时被取消；因为项目失败而导致的官司和诉讼仍然会是普遍现象；软件产品的质量仍然会很差；软件项目的生产率仍然会保持在较低的水平；软件中的安全漏洞仍然多得惊人；而软件工程文献也将会继续发布一些毫无根据的声明却不提供真实可信的数据。

其实，软件工程行业需要做的事情非常明确：

1. 度量所有可能的缺陷来源并且这些来源的材料都是用功能点描述。包括需求缺陷、设计缺陷、代码缺陷、文档缺陷以及不成功的缺陷修复所导致的次生缺陷。
2. 度量各种形式的审查、静态分析和测试的缺陷去除效率水平。
3. 建立基于软件开发活动的生产率基准，并保证覆盖从需求到产品交付过程中的所有活动；然后为软件维护和客户支持活动建立类似基准，覆盖从产品交付到产品退出市场过程。以上所有的基准都要基于功能点方法。
4. 建立经过认证的接近零缺陷的可重用材料库。
5. 大幅度提高软件安全防护方法的效率以保护软件免受病毒、间谍软件和黑客的攻击。
6. 建立软件工程专业人员的执照和行业委员会认证系统。

但是如果对于软件项目的度量方法不能做到准确并对费用进行有效考量的话，以上所有的工作都不可能完成。如果一个行业无法对其自身水准进行准确度量的话，那么这个行业就不能算作一门真正的职业。

8. 代码行度量方式在 2020 年左右的应用

如果我们对 2020 年做一个展望的话，我们需要同时考虑最好的情况和最差的情况。

最好的情况就是代码行方式的使用率大幅度降低，其降低速度甚至超过了它当初的发展速度。同时软件行业开始注重考量整个产品交付过程中的经济生产率而不仅仅是开发过程和代码行数量。为了让这种预测成为现实，我们需要提高功能点分析工具的工作速度，同时将计算功能点所需要的费用由每个功能点大约 6 美元降低到每个功能点低于 0.1 美元。这一费用标准在技术上是可行的，并且在 2009 年的今天确实是存在的。只是由于其出现时间太短因而还没有得到大范围的部署和应用。

如果上面描述的那些变化确实发生了,那么功能点方法的应用范围将会至少扩大10倍,同时我们也可以进行更多种类的经济研究。这些新种类的经济研究包括对整个软件系列进行度量(整个软件系列可能包含超过1千万个功能点),估算企业级待办事宜清单的规模并对清单中的内容进行排序(部分清单的规模可能会超过1百万个功能点),使对主要软件进行风险分析/价值分析成为日常工作并确保其结果真实可信。同时,我们还有可能对感兴趣的新兴科技进行经济分析,如敏捷开发、面向服务的架构(SOA)、软件即服务(SaaS),当然还包括总拥有成本(TCO)。

在最好的情况下,软件工程将会从一门手艺或者一种艺术形式发展成为一个真正的工程学科。对软件工程中所有活动和任务提供准确可靠的度量将会帮助我们提高大型软件应用项目的成功率。软件工程的目标应该是成为一门真正的工程学科,有行业承认的专家、行业委员会提供的认证以及项目生产率、质量和费用的准确信息。但是目前对于大型软件项目来说,失败的项目多于成功的项目,在这种情况下我们是无法实现上述目标的。

只要软件项目的质量和生产率仍然是模糊和不确定的,我们就很难实施多种回归分析并选择出真正有效的方法和工具。代码行方法已经成为了对软件项目进行经济研究和质量研究的一个主要障碍。

最坏的情况是,代码行度量方式会保持2009年的应用范围和水平。软件工程行业会继续忽略项目的经济生产率而仅仅将注意力集中在虚幻的“单位月份完成的代码行数”之类的度量标准上。在最坏的情况下,“软件工程”这个词仍然是一个矛盾体。试错法仍然在项目中占据主要地位,出现这一现象的部分原因是因为使用代码行方式时,人们无法对一些有效的工具和方法进行研究。在最坏的情况下,大型软件项目的失败或者出现灾难性的后果仍然会是普遍现象。

而功能点度量方法将会在软件项目的经济学研究、软件行业标准及基准分析中扮演一个重要的角色,但是仅仅应用在大约10%的中型软件项目中。在最坏的情况下,计算单位功能点的费用仍然很高,以至于功能点方式仍然极少在超过15 000个功能点的项目中使用。人们可能会开发出更多的变化形式来计算功能点,但是仍然会缺乏从一种方法到另一种方法的转换规则,而长期缺乏这种规则使得对于大型国际项目进行经济研究几乎是不可能的。

8.15 总结

代码行度量方式的历史为所有在软件业工作的人们敲响了警钟。这种度量方式在开始时起到了很好的作用。当时人们只有一种编程语言,并且由于编程工作的困难性导致在计算机上进行程序编写占据了整个软件项目总工作量的90%,这种度量方式是十分有效的。

但是人们逐渐开始开发出数百种编程语言,同时软件应用的开发工作开始混合使用多种编程语言(这一点直到今天都还是业内常用的做法);软件应用的规模开始扩大,从最初的不到1000行代码发展到超过1百万行代码;对于小型软件来说,编码工作仍是项目工作量的主体部分,但对于大型软件系统来说,主要的工作量转移到了缺陷去除工作以及各种形式的文档编写工作,包括需求文档、产品说明书、用户手册、测试计划等各种各样的书

面文档。

然而,代码行度量方式并没有和软件行业的这些变化保持同步。如果一个项目使用了高级语言,特别是同一个软件中使用了多种语言,我们在计算其代码行数的时候通常都会遇到问题从而导致其结果并不准确。在这种情况下,代码行方式并不能很好地进行分析和度量。同样,代码行方式也不适用于那些编码工作仅占总工作量一小部分的大型软件系统。

因此,代码行度量方式的作用变得越来越小。直到大约1985年左右的时候,这种度量方式开始对软件项目产生危害。考虑到这种度量方式在人们研究软件项目的经济效益、生产率和质量的时候带来的误解甚至错误,我们可以公平地说,如果一项研究中包含多种编程语言或者这项研究的目的是考量软件项目真实生产率,那么在多数情况下使用代码行度量方式我们可以视其为失职。

我们最终的观点是,继续使用代码行度量方式会严重阻碍软件工程行业发展的进度。它会成为软件行业从一种手艺转变成成为真正的工程学科的障碍。如果一个行业无法对其自身工作进行准确度量的话,它就很难成为一门真正的职业。

参考文献

- Barr, Michael and Anthony Massa. *Programming Embedded Systems: With C and GNU Development Tools*. Sebastopol, CA: O'Reilly Media, 2006. (中文版《嵌入式系统编程——使用C和GNU开发工具》(第二版), 王映辉、王琼芳、李军怀译, 中国电力出版社2009年5月出版)
- Beck, K. *Extreme Programming Explained: Embrace Change*. Boston, MA: Addison Wesley, 1999. (中文版《解析极限编程——拥抱变化》(原书第二版), 雷剑文、李应樵、陈振冲译, 机械工业出版社2011年9月出版)
- Bott, Frank, A. Coleman, J. Eaton, and D. Rowland. *Professional Issues in Software Engineering*, Third Edition. London and New York: Taylor & Francis, 2000.
- Cockburn, Alistair. *Agile Software Development*. Boston, MA: Addison Wesley, 2001.
- Cohen, D., M. Lindvall, & P. Costa, "An Introduction to agile methods." *Advances in Computers*. New York: Elsevier Science (2004): 1-66.
- Garms, David and David Herron. *Function Point Analysis*. Boston: Addison Wesley, 2001. (中文版《敏捷软件开发》(原书第二版), 苏敬凯译, 机械工业出版社2008年1月出版)
- Garms, David and David Herron. *Measuring the Software Process: A Practical Guide to Functional Measurement*. Englewood Cliffs, NJ: Prentice Hall, 1995.
- Glass, Robert L. *Facts and Fallacies of Software Engineering (Agile Software Development)*. Boston: Addison Wesley, 2002. (中文版《软件工程的事实与谬误》, 严亚军、龚波译, 中国电力出版社2006年1月出版)
- Hans, Professor van Vliet. *Software Engineering Principles and Practices*, Third Edition. London, New York: John Wiley & Sons, 2008.
- Highsmith, Jim. *Agile Software Development Ecosystems*. Boston, MA: Addison Wesley, 2002. (中文版《敏捷软件开发生态系统》, 姚旺生、杨鹏译, 机械工业出版社2004年1月出版)

- Humphrey, Watts. *PSP: A Self-Improvement Process for Software Engineers*. Upper Saddle River, NJ: Addison Wesley, 2005. (中文版《PSP(SM) 软件工程师的自我改进过程》, 吴超英、赵泓峰、余毅、卢祺译, 人民邮电出版社 2006 年 6 月出版)
- Humphrey, Watts. *TSP—Leading a Development Team*. Boston, MA: Addison Wesley, 2006. (中文版《TSP——领导开发团队》, 张家才、江贺、车皓阳译, 人民邮电出版社 2007 年 1 月出版)
- Hunt, Andrew and David Thomas. *The Pragmatic Programmer*. Boston, MA: Addison Wesley, 1999. (中文版《程序员修炼之道: 从小工到专家》, 马维达译, 电子工业出版社 2011 年 1 月出版)
- Jeffries, R., et al. *Extreme Programming Installed*. Boston, MA: Addison Wesley, 2001. (中文版《极限编程实施》, 袁国忠译, 人民邮电出版社 2002 年 7 月出版)
- Jones, Capers. *Applied Software Measurement*, Third Edition. New York, NY: McGraw-Hill, 2008.
- Jones, Capers. *Conflict and Litigation Between Software Clients and Developers*, Version 6. Burlington, MA: Software Productivity Research, June 2006. 54 pages.
- Jones, Capers. *Estimating Software Costs*, Second Edition. New York, NY: McGraw-Hill, 2007. (中文版《软件项目估计》, 刘从越、郝建材、申冬凯译, 电子工业出版社 2008 年 3 月出版)
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Boston, MA: Addison Wesley Longman, 2000. (中文版《软件评估: 基准测试与最佳实践》, 韩柯等译, 机械工业出版社 2003 年 4 月出版)
- Jones, Capers. "The Economics of Object-Oriented Software." *American Programmer Magazine*, October 1994: 29–35.
- Kan, Stephen H. *Metrics and Models in Software Quality Engineering*, Second Edition. Boston, MA: Addison Wesley Longman, 2003. (中文版《软件质量工程——度量与模型》(第二版), 吴明晖、应品译, 电子工业出版社 2004 年 7 月出版)
- Krutchén, Phillippe. *The Rational Unified Process—An Introduction*. Boston, MA: Addison Wesley, 2003.
- Larman, Craig & Victor Basili. "Iterative and Incremental Development—A Brief History." *IEEE Computer Society*, June 2003: 47–55.
- Love, Tom. *Object Lessons*. New York, NY: SIGS Books, 1993.
- Marciniak, John J. (Ed.) *Encyclopedia of Software Engineering*. (2 vols.) New York, NY: John Wiley & Sons, 1994.
- McConnell, Steve. *Code Complete*. Redmond, WA: Microsoft Press, 1993. (中文版《代码大全》(第二版), 金戈、汤凌、陈硕、张菲译, 电子工业出版社 2006 年 3 月出版)
- . *Software Estimation—Demystifying the Black Art*. Redmond, WA: Microsoft Press, 2006.
- Mills, H., M. Dyer, & R. Linger. "Cleanroom Software Engineering." *IEEE Software*, 4, 5 (Sept. 1987): 19–25.
- Morrison, J. Paul. *Flow-Based Programming. A New Approach to Application Development*. New York, NY: Van Nostrand Reinhold, 1994.
- Park, Robert E. *SEI-92-TR-20: Software Size Measurement: A Framework for Counting Software Source Statements*. Pittsburgh, PA: Software Engineering Institute, 1992.
- Pressman, Roger. *Software Engineering—Practitioner's Approach*, Sixth Edition. New York, NY: McGraw-

- Hill, 2005. (中文版《软件工程—实践者的研究方法》(原书第七版), 郑人杰、马素霞译, 机械工业出版社 2011 年 5 月出版)
- Putnam, Lawrence and Ware Myers. *Industrial Strength Software—Effective Management Using Measurement*. Los Alamitos, CA: IEEE Press, 1997.
- *Measures for Excellence—Reliable Software On-Time Within Budget*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall, 1992.
- Sommerville, Ian. *Software Engineering*, Seventh Edition. Boston, MA: Addison Wesley, 2004. (中文版《软件工程》(原书第九版), 程成译, 机械工业出版社 2011 年 5 月出版)
- Stapleton, J. *DSDM—Dynamic System Development Method in Practice*. Boston, MA: Addison Wesley, 1997.
- Stephens M. and D. Rosenberg. *Extreme Programming Refactored: The Case Against XP*. Berkeley, CA: Apress L.P., 2003. (中文版《重构极限编程——XP 的实践与反思》, 汪丰、赵浩译, 清华大学出版社 2005 年 6 月出版)

软件质量：软件工程成功的关键

9.1 引言

自 1979 年以来，美国的整体软件质量平均水平几乎没有任何变化。即便如此，软件行业的某些公司仍然在软件质量方面取得了显著进步。取得这些进步的是那些对软件质量进行严格度量的公司，它们以这样一种方式定义软件质量：软件质量的预测和度量均并非不可能。

这些公司还使用全套的软件缺陷去除活动，其中包括正式审查、静态分析以及各种软件测试活动。一旦认识到软件质量对软件工程成功的重要性，他们还会使用诸多缺陷预防方法，如联合应用设计（JAD）；和专注于软件质量的开发方法，如团队软件过程（TSP）。

从历史上看，大型软件项目在寻找和修复缺陷上花费的时间和精力比任何其他软件活动都要多。由于软件缺陷去除效率平均只有大约 85%，因此，软件维护的主要工作是发现和修复交付给客户之后的缺陷。

由于软件项目成本的 30% 到 50% 都花在了发现和修复软件缺陷上，软件开发和软件维护中的缺陷去除活动费用构成了软件项目总拥有成本（TCO）的主要部分。

过高的缺陷数目是软件项目进度落后及预算超支的主要原因之一，它拖慢了测试速度，直接导致项目进度延误、成本超支。

当软件项目被取消并最终因违反合同而闹上法庭时，几乎所有诉讼案件都与缺陷数目过多、缺陷去除不足以及软件质量度量糟糕有关。

软件缺陷去除成本已成为过去 50 年所有主要软件项目的首要成本因素，但人们仍然对软件质量了解得却如此之少，实在令人惊讶！

目前存在很多关于软件质量和测试的书籍，但几乎没有包含如下基本主题的可靠、可信的定量数据：

1. 具体新软件应用中会出现多少缺陷？
2. 遗留软件应用中可能会出现多少缺陷？
3. 如何预测和度量软件质量？
4. ISO 标准在质量改进上到底多么有效？
5. 软件质量保证组织在质量改进上到底多么有效？
6. 用于质量改进的软件质量保证认证到底多么有效？

7. 六西格玛在质量改进上到底多么有效?
8. 质量功能展开 (QFD) 在质量改进上到底多么有效?
9. CMMI 较高等级 (通常是 4 ~ 5 级——译者注) 在质量改进上到底多么有效?
10. 各种形式的敏捷开发在质量改进上多么有效?
11. Rational 统一过程 (RUP) 在质量改进上多么有效?
12. 团队软件过程 (TSP) 在质量改进上多么有效?
13. ITIL 方法在质量改进上多么有效?
14. 面向服务软件架构 (SOA) 在质量改进上多么有效?
15. 经过认证的可重用组件在质量改进上多么有效?
16. 正式审查可以清除多少缺陷?
17. 静态分析可以清除多少缺陷?
18. 软件测试可以清除多少缺陷?
19. 需要多少种不同类型的测试?
20. 需要多少测试人员?
21. 与开发者相比, 测试专家多大程度有效?
22. 自动化软件测试多么有效?
23. 对于不同规模大小的应用, 需要多少测试用例?
24. 测试认证在性能改进上多么有效?
25. 有多少缺陷修复会导致新缺陷产生?
26. 多少缺陷会被交付给用户?
27. 改进软件质量到底花费几何?
28. 改进软件质量需要多长时间?
29. 从软件质量改进中我们能节省多少时间和成本?
30. 良好软件质量给我们带来的投资回报 (ROI) 是多少?

本章的目的就是展示质量保证活动的每种主要形式、审查阶段、静态分析及测试阶段等在软件应用交付缺陷水平上的量化结果。

缺陷去除活动包括“个人”和“公共”两种形式。个人形式的软件缺陷去除活动包括桌面检查、静态分析和单元测试, 这已在第 8 章中讨论过, 因为这些方法专注于代码缺陷而第 8 章正好讨论编程和代码开发。

公共形式的缺陷去除活动包括正式审查、静态分析 (由编写代码的软件工程师之外的人来执行) 以及由测试专家而不是开发者自己执行的各种测试方法。

所有个人形式和公共形式的软件缺陷去除方法都很重要, 但是很难从个人缺陷去除方法中获得相应数据, 因为这些缺陷去除活动通常是在没有其他人参与的情况下进行的。正如第 8 章所指出的, IBM 使用志愿者来记录个人缺陷去除活动中发现的缺陷数据。一些开发方法, 如 Watts Humphrey 的团队软件过程 (TSP) 和个体软件过程 (PSP) 也会记录个人缺陷去除中的有关数据。

本章还将解释如何预测可能出现的错误或缺陷数目, 以及如何预测缺陷去除的效率水

平。这不仅包括代码错误，还包括需求、设计和文档中需要预测的错误或缺陷。此外，还需要预测缺陷修复过程中意外引入的新缺陷，常称为“不良修复”（bad fix）。最后，测试用例本身也可能含有缺陷或错误，这些也需要预测。

本章将会讨论质量度量的最好方法，以及对危险的度量指标，如“平均缺陷成本”和“代码行”给出警告，这些度量指标扭曲了软件质量度量结果，掩盖了软件质量的真实情况。本章将讨论如下软件质量的几个关键主题：

- 软件质量定义
- 软件质量预测
- 软件质量度量
- 软件缺陷预防
- 软件缺陷去除
- 软件质量专家
- 软件质量的经济价值

软件质量是软件工程成功的关键。无论是开发期间还是发布之后，软件工程长期以来饱受过高缺陷数量的困扰。现在已有众多技术可以降低软件缺陷数目，从而极大地改进软件质量。

仔细地规划和选择缺陷预防和缺陷去除活动的有效组合可以缩短软件开发进程、降低软件开发成本、显著地减少软件维护和客户支持成本，同时提高客户满意度和员工士气。在当前任何软件过程改进方式中，软件质量改进是投资回报最高的一个。

随着经济衰退的继续，每个公司都渴望降低软件开发和维护成本。软件质量改进将比其他任何可用的技术更有助于提高软件的经济性。

9.2 软件质量定义

人们很难给出一个好的软件质量定义。目前，软件文献中已公开发表了许多不同的软件质量定义。不幸的是，某些已公布的质量定义要么太过抽象，要么偏离现实。切实可行的软件质量定义需要具备以下 6 个基本特征：

1. 软件应用运行之前，软件质量应该是可预测的。
2. 软件质量需要囊括所有可交付物而不仅仅只包括代码。
3. 软件质量应该在开发期间可度量。
4. 软件质量应该在软件发布给客户之后仍然可度量。
5. 对客户来说，软件质量应该是显而易见的，且获得客户认可。
6. 在发布之后的维护阶段，软件质量应该仍然是持续的。

下面是软件质量的一些定义，并解释为什么某些定义不符合上述所列的 6 条标准。

9.2.1 质量定义 1：软件质量意味着最终软件产品符合软件需求

这个定义有几个问题。主要问题是，在一般的软件项目中，软件需求的错误和缺陷数

量较多,而且大多都比较严重。在软件项目中,需求缺陷通常占到全部软件缺陷的20%左右,并直接或间接导致超过35%的高严重性缺陷。

将软件质量定义为最终开发出来的软件产品满足用户需求这一主要错误来源是一种循环论证,因而必须认识到这个定义有问题、行不通。显然,一个可行的质量定义必须能够处理需求本身的错误。

不要忘记著名的“千年虫”(Y2K)问题^①,它起源于一个特殊的用户需求故而它并不是代码缺陷。许多软件工程师曾警告客户和管理人员,两位数字的日期字段限制会导致很多问题,但他们的警告要么被忽略,要么被断然拒绝。

笔者曾在一个诉讼案件中短期地担任专家证人。在该诉讼案件中,一家公司试图起诉其软件外包供应商在根据合同开发的软件中使用两位数字的日期字段。在举证阶段,各种证据显示软件外包供应商曾警告过该公司两位数字的日期字段比较危险,但该公司拒绝了软件外包供应商的建议并坚持两位数字的日期字段,这导致其应用中被引入了“千年虫”问题。事实上,该公司自己的内部标准强制规定了两两位数字的日期字段。不用说,当明显证明他们自己才是这个问题的产生根源时,该公司只好撤销了诉讼。这个案子说明,“用户需求”经常饱含错误,有时软件需求甚至是危险或“有毒”(toxic)的。

该案件还说明了另一点。无论是原告的高层管理人员还是法律部门都不知道“千年虫”问题是由他们自己的政策和实践引起的。很明显,当公司高管不能很好地理解诸如此类问题时,就需要来自公司顶层更好的软件治理。

使用自经济衰退以来的现代语言讲,软件开发安全地符合软件需求之前,有必要首先去除“毒性需求”。“软件产品符合用户需求”这样的质量定义并不能随着时间推移而带来任何重大的质量改进。2009年与1979年相比并没有更多的需求得以满足,而软件产品的质量也没有明显的提高。

如果软件工程要真正成为一个行业而不是一种艺术形式(art form),软件工程师们有责任以细致、精确、有效的方式帮助客户定义需求。坚持使用有效的需求方法,如联合应用设计(JAD)、质量功能展开(QFD)及需求审查是每一个职业软件工程师的职责。

大量软件质量文献在识别软件需求上都比较消极,且常常给出错误假设:用户可以100%有效地识别需求。这个假设极其危险,事实上,用户需求从来没有完整过且常常满含错误。一个软件项目要想成功,需要以专业的方式来收集和分析需求,而软件工程应该正是知道如何很好地做到这一点的行业。

坚持使用正确的需求方法应该是软件工程师的职责。这些正确的需求方法包括联合应用设计(JAD)、质量功能展开(QFD)以及需求审查。也可能推荐使用其他有利于定义需求的方法,如客户参与或用例。用户自己不是软件工程师,软件行业不能期望用户知道分析和表达需求的最佳方法。软件工程团队有责任确保以最先进的方法收集和分析软件需求。

一旦用户需求收集和分析完成,符合这些需求应该是自然而然的事情。然而,在安全

^① Y2K problem, 也称“计算机2000年问题”。为节省存储空间,计算机中的日期存储仅采用了后两位数字。这样,从1999年到2000年,日期将从99变为00。计算机软件很可能将日期认为是1900而不是2000。这样诸如银行、财务等日期敏感软件将产生严重错误。——译者注

有效地开发出满足这些需求的软件之前，危险和“有毒”需求必须被清理出去，应该向用户指出哪些需求是过度的和不必要的需求，应该识别和量化会导致需求蔓延的任何潜在不足。用户自己需要来自软件工程团队的专业帮助，软件工程团队不应该在需求收集和分析上被动地袖手旁观。

不幸的是，需求缺陷无法靠普通测试去除掉。如果无法防止需求缺陷的发生，或者无法通过正式审查去除掉需求缺陷，由这些需求构建出的测试用例将“接受”这些需求缺陷而无法发现它们。（这就是为什么这么多年的软件测试从未发现和去除“千年虫”问题。）

该软件质量定义的第二个问题是，软件质量在开发期间无法预测。软件产品是否满足初始需求可以在软件开发完成之后通过测试等手段进行验证和度量，但就有效成本控制而言，如果软件有问题，则为时已晚。

该质量定义的第三个问题是，对于全新的创新类应用，除初始创新者之外可能还没有任何用户。这方面的例子，考虑一下成功软件创新的历史，如 APL 编程语言、第一个电子表格以及后来成为 Google 的早期 Web 搜索引擎。

这些创新应用都是创新者为解决某个他们自己想要解决的问题而创造出来的，不是根据正常的“用户需求”概念而开发的。直到开发出原型，其他人很少知道这些创新会有怎样的实际价值。因此，这些新应用被披露给公众之前，“用户需求”完全与全新创新应用没有关系。

事实上，软件需求在后续设计和编码阶段每月以 1% 到超过 2% 的可度量速率增长和变化。很明显，完全理解用户需求是一个很困难的任务。

软件需求非常重要，但“毒性”需求、残缺需求以及过度需求的掺杂，使得诸如“软件质量意味着软件产品符合用户需求”这样过分简化的质量定义对软件工程并无益处，甚至有害。

9.2.2 质量定义 2：“软件质量意味着软件产品具有很好的可靠性、可移植性及诸多其他特性”

该方法把软件质量定义为软件产品具有一组不同类型的“特性”。这种定义方式的问题是，很多特性既不能在实现它们之前对其进行预测，又不能在实现它们之时很容易地对其进行度量。

虽然大多数特性对软件应用来说是有用的属性，但当我们考虑某些物理设备，如汽车或者烤面包机的时候，某些性能似乎又与质量无关。例如，“可移植性”对软件厂商来说可能是有用的，但在大多数用户眼里它跟软件质量似乎没有关系。

使用各种特性来定义软件质量并不会随着时间推移促使软件质量有所提高。2009 年，软件行业在很多特性上做得并不比 1979 年时更好。使用自经济衰退以来的时髦语言讲，这种定义的很多特性都属于“次贷式”定义，它们并不能预防任何严重的质量缺陷。实际上，使用各种特性来定义软件质量而不是专注于缺陷预防和缺陷去除会拖慢软件质量控制的进程。

当使用软件质量的这个定义时，我们能列举的许多特性如下所示（以字母顺序排序）：

1. 可扩展性 (Augmentability)
2. 兼容性 (Compatibility)
3. 可扩张性 (Expandability)
4. 灵活性 (Flexibility)
5. 互操作性 (Interoperability)
6. 可维护性 (maintainability)
7. 可管理性 (Manageability)
8. 可修改性 (Modifiability)
9. 可操作性 (Operability)
10. 可移植性 (Portability)
11. 可靠性 (Reliability)
12. 可伸缩性 (Scalability)
13. 可生存性 (Survivability)
14. 可理解性 (Understandability)
15. 可用性 (Usability)
16. 可测试性 (Testability)
17. 可追踪性 (Traceability)
18. 可验证性 (Verifiability)

上述列表中, 在用户看来, 似乎仅有几个如“可靠性”和“可测试性”与质量有关。其他一些特性则不是令人费解(如“可生存性”)就是有些用处但与质量无关(如“可移植性”)。软件厂商或开发团队可能对某些特性兴趣浓厚, 但客户则毫无兴趣(如“可维护性”)。

这些特性似乎在学术研究上很有价值, 但它们并不能真正解决现实世界中某些困扰客户的质量问题。例如, 当客户指出一个软件错误或者软件行为不正常时, 没有哪个特性可以解决“客户联系到客户支持人员以获得帮助的难易程度”这一问题, 也没有哪个特性涉及修复缺陷并向用户提供修复补丁的响应速度问题。

在用户心目中, 新的信息技术基础架构库 (ITIL) 在处理诸如客户支持、事件管理和缺陷修复时间间隔等软件质量问题方面所做的工作要比标准文献好得多。

更严重的是, 上述特性列表忽视了当软件最终被发布给客户时对软件质量有重大影响的 2 个主要主题: (1) 潜在缺陷; (2) 缺陷去除效率水平。

术语“潜在缺陷”是指设计和构建软件应用时可能会出现的所有缺陷总数。潜在缺陷包括需求、设计、编码、用户文档中的错误或缺陷以及不良修复或次生缺陷。术语“缺陷去除效率”指以任意顺序进行的审查、静态分析和测试阶段发现并修复的缺陷占潜在缺陷的百分比。

为达到客户看来可以接受的质量水平, 人们需要较低的潜在缺陷和较高的缺陷去除效率 (高于 95%), 二者相辅相成。当前美国软件质量的平均水平是每功能点大概 5 个潜在缺陷和 85% 的缺陷去除效率水平。由此总交付缺陷为每功能点大约 0.75 个缺陷, 笔者认为这

既不专业又不可接受。

要成为一个真正的工程行业，软件工程需要认真对待，以将潜在缺陷降低到低于每个功能点 2.5 个缺陷以及缺陷去除效率水平提高到平均 95% 以上。这种组合的结果是，交付缺陷总数降低到每个功能点只有 0.125 个缺陷，只是现今平均水平的 1/6。2009 年的今天，达到或超过这个质量水平完全有可能，但却鲜有实现。

软件项目没有广泛实现本来应该达到的良好质量水平，原因之一是软件工程活动往往集中精力于实现各种特性而非缺陷度量和缺陷去除效率，这导致了缺陷去除活动的严重不足和失败。换句话说，用各种特性来定义软件质量会严重干扰发生软件缺陷根本原因和预防方法以及最好的软件缺陷去除方法的深入研究。

我们可以在软件外包协议中包含明确的潜在缺陷水平和缺陷去除效率水平条款。这可能会比当前的质量承包实践更加有效，因为目前的质量承包实践中往往不存在潜在缺陷水平和缺陷去除效率水平要求或者仅仅要求保持某个确定的 CMMI 等级。

如果软件带着过多的缺陷发布出去，由此导致停止运行、行为异常或者运行缓慢，那么人们很快就会发现，大多数的上述特性都显得不太重要了。

对软件用户而言，已发布软件中的缺陷数量往往是至关重要的质量问题。另一个重要问题是，一旦缺陷被曝光，软件供应商将会采取什么样的纠正措施。这些问题引出了第三个与软件质量更为紧密相关的软件质量定义。

9.2.3 质量定义 3：质量就是软件产品中不存在会导致软件停止工作或行为不正确的缺陷

软件缺陷是软件的漏洞或者错误，它会导致软件要么停止运行，要么产生无效或不可接受的结果。使用 IBM 的严重性等级，缺陷具有以下 4 个严重性级别：

1. 严重性 1 级意味着软件应用根本就不工作。
2. 严重性 2 级表示主要功能失效或产生不正确结果。
3. 严重性 3 级表示有些小问题或次要功能不能正常工作。
4. 严重性 4 级意味着只是个不影响任何操作的微小问题。

由于是人为地给每个缺陷分配严重性等级，上述缺陷严重性等级有些主观。依照 IBM 的模式，当首次报告一个缺陷时，基于报告该缺陷的客户或用户所描述的症状，会给这个缺陷分配一个初始严重性等级。然后，最终的严重性等级由修复该缺陷的团队在修复该缺陷时确定。

笔者青睐这个质量定义有几个原因。首先，缺陷发生之前可以预测而缺陷出现之时又可以度量。其次，许多软件应用的客户满意度调查似乎与交付缺陷水平而不是任何其他因素更加紧密相关。最后，很多特性因素也与有没有软件缺陷相互关联。例如，可靠性精确地与在软件中发现的缺陷数目相关。易用性、可测试性、可追踪性和可验证性也与软件缺陷水平间接相关。

度量软件缺陷数量和缺陷严重性等级，然后通过缺陷预防和缺陷去除活动相结合采取有效步骤去减少软件中存在的缺陷，此乃软件工程成功的关键。

该软件质量定义确实会随着时间的推移提高软件的质量。那些认真度量潜在缺陷、缺陷去除效率水平和交付缺陷数量的公司已在这些因素上有了很大程度的提高。该质量定义支持流程改进、质量预测、质量度量和以调查数据来衡量的客户满意度。

所以,那些进行质量度量的公司,如IBM、Dové Technologies及AT&T,已经在质量控制方面取得了不小进展。还有,将缺陷跟踪和缺陷报告进行集成的开发方法(如团队软件过程(TSP))也已在减少交付缺陷方面做出了重大进步。对于那些已在其缺陷去除工具套件中添加了静态分析的开源应用,同样也进步不小。

缺陷及其去除效率量已被用于验证正式审查的有效性、展示静态分析方法的影响及对超过15种软件测试方法进行微调。那些具有主观性的措施则没有能力处理这些问题。

每一位软件工程师和每一位软件项目经理都应该接受培训,学习软件缺陷预测、软件缺陷度量、软件缺陷预防和软件缺陷去除的方法。如果没有有效的质量和缺陷控制知识,软件工程就是一个骗局。

笔者建议的完整质量定义包括以下9个方面:

1. 质量意味着软件被部署时的低缺陷水平,理想地接近零缺陷。
2. 质量意味着高可靠性,或者没有宕机、行为怪异、非期望结果或性能迟缓等现象。
3. 质量意味着当对软件应用及其功能进行调查时的高用户满意度水平。
4. 质量意味着满足大多数客户或用户正常运作需要的功能集合。
5. 质量意味着良好的代码结构和清晰适度的注释密度,当试图修复旧缺陷时最大限度地减少不良修复和意外引入新错误,也有利于软件添加新的功能。
6. 质量意味着当问题发生时有效的客户支持,客户联系支持团队和获取帮助的难度最小。
7. 质量意味着快速修复已知缺陷,特别是高严重性缺陷。
8. 质量意味着支持由软件开发者提供给软件客户有意义的担保和保证。
9. 有效质量定义应该促使质量不断提高。这意味着软件质量需要足够严格的定义,以使质量提高和下滑都可识别,对平均水平也一样。如果质量定义不能反映软件质量的变化或改善,那它的价值非常有限。

以上这些质量涵义中的第6项、第7项、第8项和第9项,除了新的ITIL书籍,当前软件质量文献中鲜有涵盖。不幸的是,ITIL所涵盖的内容只适用于公司内部软件,因而商业软件厂商完全忽略了它。

从软件用户和客户的角度看,“没有软件缺陷”的软件质量定义结合其他补充主题,如方便的客户支持和快捷的维护速度,体现了软件质量的本质。

考虑一下将本章所讨论的3个质量定义应用到某个知名软件产品,比如微软的Windows Vista又会怎么样?选择Windows Vista作为例子是因为这个产品是最著名的大型软件应用之一,因而是用于尝试各种质量定义的良好测试床^①。

9.2.4 将定义1应用到Vista:质量意味着软件产品符合用户需求

尽管可能在某些方面上使用了焦点小组,既然没有普通客户被问到在操作系统中他们

^① 意指测试对象或实验环境。——译者注

需要什么功能，那么第一个定义很难适用于 Vista。

如果你将 Windows Vista 与 Windows XP、Leopard^①或者 Linux 进行对比就会发现，它似乎包含了过多的特色和功能，其中许多功能既不是用户请求的，也不是大多数用户曾经用过的。软件工程文献没有很好地讲述甚至根本没有涵盖的一个主题是，应用软件过度填充了不必要的或无用的功能。

大多数人都知道，普通需求通常会忽略约 20% 用户希望的功能。然而，没有多少人知道由如微软、赛门铁克、CA 及类似公司生产的商业软件，超过 40% 的功能可能是客户不需要或从来不用。

无论是模仿竞争对手所做的，还是试图依靠提供数以百计昂贵而边缘化但其竞争者又无法模仿的功能来超越那些较小的竞争者，功能填充（feature stuffing）本质上是一种竞争性的举措。无论何种情况，功能填充不是对用户需求令人满意的举措。

更进一步讲，操作系统用户真正欣赏的某些基本功能，比如安全性和运行性能，则在 Vista 上没有特别好地体现出来。

最根本的是，对于像 Vista 这样拥有超过 100 万用户的应用软件来说，把“软件产品满足用户需求”定义为软件质量是毫无用处的，因为不可能知道如此大范围的用户真正想要的和不需要的功能。

此外，用户很少能够有效地表达自己的真实需求，因此帮助用户认真而精确地定义需求是专业软件工程师的职责。太多的软件文献假设软件工程师只需被动观察用户需求，而实际上，软件工程师应当扮演需求分析的主动角色，就像诊断患者身体状况以开出有效疗法处方的内科医生那样的角色。

内科医生不会只是被动地询问病人哪里有疾病以及病人想要服用什么样的药。作为软件工程师，我们的工作既是拥有关于有效需求收集和分析方法（即医疗诊断测试）的专业知识，也知道什么类型的应用可以为用户需求提供有效的“疗法”。

被动地等待用户定义需求而没有协助用户使用联合应用设计（JAD）或质量功能展开（QFD）或者遗留应用的数据挖掘方法，这在部分软件工程社区被认为是不专业的。用户并没有接受过任何需求定义的培训，所以软件工程团队需要上一步承担起协助用户定义需求的任务。

9.2.5 将定义 2 应用到 Vista：质量意味着软件产品满足各种“特性”要求

当把 Vista 的功能和早先所列的各种“特性”术语列表做匹配来判断 Vista 质量的时候，就会看到应用这样一个列表实际上多么抽象和困难。

1. 可扩展性	含糊不清，难以适用于 Vista	5. 互操作性	含糊不清，难以适用于 Vista
2. 兼容性	较差；许多旧应用无法工作	6. 可维护性	对用户未知，但可能对 Vista 来说较差
3. 可扩张性	适用于 Vista 且相当不错	7. 可管理性	含糊不清，难以适用于 Vista
4. 灵活性	含糊不清，难以适用于 Vista	8. 可修改性	对用户未知，但可能对 Vista 来说较差

① 苹果公司于 2007 年秋季推出的新版操作系统 Mac OS X 10.5 版的代号。——译者注

(续)

9. 可操作性	含糊不清, 难以适用于 Vista	14. 可理解性	较差
10. 可移植性	较差	15. 可用性	宣称较好, 但仍有问题
11. 可靠性	起初较差, 但后来有所提高	16. 可测试性	较差: 复杂度太高
12. 可伸缩性	边缘功能	17. 可追踪性	较差: 复杂度太高
13. 可生存性	含糊不清, 难以适用于 Vista	18. 可验证性	含糊不清, 难以适用于 Vista

最基本的是, 超过一半的“特性”含糊不清或者很难适用于 Vista 或任何其他商业软件。可以应用于 Vista 的“特性”中, 除了可扩展性和可用性之外, 似乎没有任何一个真正满足。

许多“特性”术语既无法预测又不能度量。更糟糕的是, 即使它们可以被预测和度量, 在严格的质量控制方面, 它们也只是边缘收益 (marginal interest)。

9.2.6 将定义 3 应用到 Vista: 质量意味着软件没有缺陷, 再加上各种配套因素

对任何应用软件, 能够也应该统计发布后的缺陷。其他相关主题 (如报告缺陷的容易性和修复缺陷的速度) 也应该加以度量。

不幸的是, 对于商业软件, 并非所有上述 9 个主题都可以进行评估。微软以及许多其他软件厂商并没有公布不良修复注入缺陷的数据, 甚至没有公布已报告的缺陷总数。然而, 我们还是可以凭借期刊文章和有限的微软数据来对 8 个因素中的 6 个进行评估。

1. 尽管微软不会提供由用户发现和报告缺陷的精确数字, 但我们还是知道 Vista 是带着成千上万的软件缺陷发布出去的。

2. 起初 Vista 并不是非常可靠的, 但经过大约一年使用之后达到了可以接受的可靠程度。微软没有报告平均无故障时间或者其他可靠性度量数据。

3. 对比 XP, Vista 从未达到高用户满意度水平。主要不满意之处包括打印机驱动程序缺乏、与旧应用的兼容性较差、过度资源占用以及在任何没有高端计算机芯片和大内存的系统上性能表现低迷。

4. 正如客户调查所指出的, 除了过多安全漏洞外, Vista 的功能非常丰富。

5. 微软不发布不良修复注入缺陷或已报告缺陷数量的统计信息, 故一般公众无法获知本项因素的具体信息。

6. 微软的客户支持严重不足且很难访问和使用。这也是很多软件厂商的通病。

7. 一些已知缺陷遗留在微软 Vista 中好几年。微软在缺陷修复速度上勉强及格。

8. 目前 Vista (或者其他的商业应用) 尚无有效的质量担保。除了更换有缺陷的光盘外, 微软的终端用户许可协议 (end-user license agreement, EULA) 使其免除了任何质量方面的法律责任。

9. 2009 年, 微软的新一代操作系统还未发布, 因而不可能知道微软是否使用了能生产出比 Vista 质量更好的软件方法。然而, 微软确实有大量的内部缺陷跟踪和质量保证方法, 希望新一代操作系统的质量会更好。随着时间的推移微软在质量方面已表现出了一些进步。

基于以上 9 个因素的分析模式，在以上任何质量定义下都不能说 Vista 是个高质量的应用。上述 3 个质量定义中，定义质量为“软件产品满足用户需求”几乎不可能在 Vista 上使用，因为在成百万用户的情况下，没有人能定义出每个用户都想要的功能集合。

第二个质量定义把质量定义为软件产品具有一系列的不同类型“特性”，也很难应用于 Vista，而且很多特性跟质量无关。这些特性可能对小型内部应用有些用处，但对商业软件尤其没有帮助。而且，在任何“特性”里没有发现许多关键质量主题如客户支持和维护修复次数等。

第三个定义集中在缺陷、客户支持、缺陷修复和更好的质量担保，似乎是与质量最相关的。第三个定义还具有的优势是质量是可预测和可度量的，而前两个定义都缺乏这两点。

鉴于商业软件的高成本、最低限度或根本无用的质量担保、商业软件供应商提供的差劲的客户支持，笔者赞成强制性的缺陷报告，要求如微软等商业厂商收集并发布用户所报告缺陷的数据，按缺陷严重性等级排序。

对于很多影响人类生命或安全的产品，如医药、飞机发动机、汽车以及许多其他消费类产品，强制性缺陷报告已经成为一项要求。强制性的业务和财务信息报告也是必需的。软件在很多关键方面影响着人类的生活和安全，也在很多关键方面影响着商业运营，但到目前为止，由于缺少已发布软件缺陷水平度和报告的任何强制性要求，软件行业无法就软件质量进行认真深入的研究。

令人吃惊的是，开源软件社区似乎在度量和报告软件缺陷方面更超前于老牌的商业软件厂商。许多开源公司已把缺陷跟踪和静态分析工具添加到他们的质量武器库中并使客户可以获取质量数据，而这些数据在许多商业软件厂商那里是无法获得的。

笔者也赞成类似于汽车行业伪劣商品赔偿法的商业软件“伪劣商品赔偿法”（lemon law）。当严重缺陷发生了而软件供应商在做出了有诚意的努力去解决该问题却始终无法修复该严重缺陷时，应当要求软件供应商退还购买或租赁肇事软件应用的全部费用。

“伪劣商品赔偿法”的形式也可能被应用于外包合同，除非外包失败的诉讼已提供了不能用于商业软件供应商的救济措施，因为商业软件供应商单方面的 EULA 协议免除了除更换软件介质以外的任何质量责任。

毫无疑问，软件供应商将会反对强制性的缺陷跟踪以及伪劣商品赔偿法。但是精明和有远见的厂商很快就会觉察到，这两个主题为那些知道如何控制质量的软件公司提供了显著的竞争优势。既然高质量软件也更便宜、开发更快速以及比那些漏洞百出的软件具有更低的维护成本，因而这对精明的供应商具有更加重要的经济优势。

据笔者推测，软件供应商的强制缺陷报告与伪劣商品赔偿法相结合，也许在 20 年内，将影响软件质量每 5 年提高 50% 左右。

人们需要比过去更加认真地对待软件质量。当前，经济衰退还在不断扩大，更好的软件质量控制是降低软件成本最有效的战略举措之一。但是有效质量控制依赖于更完善的质量度量措施和缺陷预防与缺陷去除活动行之有效的结合。

今天,软件质量预测、软件质量度量、更好的软件缺陷预防以及更好的软件缺陷去除方法是推动软件工程进入真正工程行业地位而非一门手艺(craft)或艺术形式(art form)的必由之路。

9.2.7 定义和预测软件缺陷

如果交付缺陷是软件的主要质量问题,那么知道是什么原因导致了这些缺陷就至关重要,这样我们就可以在软件交付之前预防这些缺陷出现或者出现后去除掉它们。

软件质量文献包含了大量关于缺陷术语的迂腐争吵,如“故障”(fault)、“错误”(error)、“问题”(bug)、“缺陷”(defect)以及许多其他术语。本书中,如果软件因为其代码错误而停止工作、不加载数据、运行不正常或者产生错误结果,那这就称为一个“缺陷”^①。(笔者在之前出版的14本书和超过30篇期刊文章中都用过这个定义,笔者始于1978年首次使用这个定义。)

然而,在现代世界,一组相同问题的发生可能不是开发者或代码的原因。被注入了病毒或间谍软件的软件也可能会停止工作、拒绝加载、运行异常以及产生不正确结果。现今世界,有些报告的缺陷可能是由外部攻击导致的。

虽然成功的攻击确实意味着软件含有安全漏洞,但是来自黑客对软件的攻击与由自己造成的缺陷迥然不同。

在本书和笔者之前的书中,软件缺陷具有5个主要来源点:

1. 需求
2. 设计
3. 代码
4. 用户文档
5. 不良修复(由于修复旧缺陷而引入了新的缺陷)

因为笔者开始研究软件质量时还在IBM工作,故在本书及笔者以前的书中使用IBM的严重性等级进行缺陷严重性级别分类,共有4个严重性等级:

- 严重性1: 软件根本不工作。
- 严重性2: 主要功能失效或行为不正确。
- 严重性3: 微小功能失效或行为不正确。
- 严重性4: 不影响正常操作的微小错误。

目前还存在很多其他的软件缺陷严重性等级分类方法,但这4个等级最常见。由于IBM在20世纪60年代就已引入,所以它们已成为一个事实标准。

软件缺陷有7种原因,主要包括:

① 术语 fault、error、bug 及 defect 在不同的文献中有不同翻译,其具体涵义在质量文献中也有诸多不同解释。本书原著中并不特指哪个问题属于哪个术语范畴,如果没有特别说明,均使用 defect 一词指代各种软件问题。本书在翻译中亦将之翻译为“缺陷”而不做特别指代。——译者注

遗漏类错误 (Errors of omission)	意外地遗漏掉某些需要的功能
功能类错误 (Errors of commission)	某些需要的功能实现不正确
歧义类错误 (Errors of ambiguity)	事情可以用好几种方式加以解释
性能类错误 (Errors of performance)	某些例程太慢而无法使用
安全类错误 (Errors of security)	安全漏洞允许外部攻击
过剩类错误 (Errors of excess)	软件中充斥着无关代码或不必要功能
去除不佳类错误 (Errors of poor removal)	本应该轻易发现而未发现的缺陷

对于不同的可交付物，这 7 个原因发生的频率不尽相同。对于诸如需求和设计等书面文档，歧义类错误最常见，其次是遗漏类错误。对于软件源代码，功能类错误最常见，其次是性能类错误和安全类错误。

第 7 个类别——去除不佳类错误——需要从根本原因上分析来加以鉴别。其含义是，该类缺陷既不是不明显的也不是很难发现，但是由于测试用例没有覆盖到相应代码段或者不完整审查的疏忽而遗漏掉了。

从某种意义上说，所有的交付缺陷都可能被视为没有及时发现并去除掉的漏网之鱼。但是，弄明白正式审查、静态分析和测试等各种各样的缺陷预防方法和缺陷去除方法为什么会遗漏这些显而易见的软件缺陷非常重要。这一类缺陷不是不容易发现的软件错误，它们本应该在软件发布之前被发现并去除掉但由于某些原因而被遗漏了，最终随着软件交付到了客户手中。

这里包括去除不佳类错误的主要原因是为了鼓励更多地学习和研究各种缺陷去除活动的有效性。人们需要更多关于正式审查、静态分析、自动化测试以及各种形式手工测试的缺陷去除效率水平的可靠数据。

缺陷来源、缺陷严重性和缺陷原因相互组合，提供了一个用于缺陷分类统计分析或根本原因分析的有用分类学方法。例如，本章早些时候提到的“千年虫”问题。就其最常见表现，使用刚刚讨论过的分类学方法，千年虫问题可能有如下描述：

注意，既然有时一个软件缺陷背后有多个问题存在，那么这个分类学方法允许使用首要因素和次要因素。

来源：	需求
严重性：	严重性 2 主要功能失效
首要原因：	功能类错误
次要原因：	去除不佳类错误

还要注意到，千年虫问题并不是在每一个应用中都有相同的严重性。数百个软件应用中千年虫问题的近似分布表明，15% 的软件停止了工作，其严重性级别为 1；严重性级别为 2 的问题占 50% 左右；严重性级别为 3 的问题占 25% 左右；抽样中有 10% 的应用软件没有操作结果。

要想知道一个缺陷的来源需要做一些研究。大多数缺陷的最初发现都是因为代码停止了工作或产生了飘忽不定的结果。但重要的是，要知道缺陷的真正原因是不是上游的问题（如需求或设计问题）。根本原因分析可以发现软件缺陷的真实原因。

上述分类方法还应该包括一些其他因素以便于对缺陷进行跟踪。这些因素包括报告的缺陷是有效缺陷还是无效缺陷。（无效缺陷很常见且相当昂贵，因为无论如何仍然需要分析它们并给以响应。）另一个因素是，一个缺陷报告是新的、独一无二的，还是仅仅是以前缺

陷报告的重复。对于软件测试和静态分析,分类“误报”(false positives)也需要包含在上述分类方法内。误报是代码段的错误识别,起初似乎不正确,但后来的深入调查表明实际上它是正确的。

第三个因素涉及修复团队是否能在他们自己的系统上重现相同问题,或者缺陷是否是由客户系统独特的配置所导致。如果缺陷不能修复,既然要解决该问题需要收集额外信息,它们被 IBM 称为待定缺陷。

添加这些额外主题到千年虫问题的例子中,产生如下扩展的分类:

当计算或预测软件缺陷时,有个标准度量指标来归一化结果是很有用的。正如第 5 章所讨论的,软件行业当前至少存在 10 个用于这种归一化指标的候选项,包括功能点、用户故事点、用例点、代码行,等等。

来源:	需求
有效性:	有效缺陷报告
唯一性:	重复(这个问题被报告了上百万次)
严重性:	严重性 2 主要功能失效
首要原因:	功能类错误
次要原因:	去除不佳类错误

在本书和笔者之前的书中,使用国际功能点用户组定义的功能点指标来量化和归一化缺陷和生产力数据。

使用 IFPUG 功能点有几个原因。就软件缺陷度量而言,最重要的原因是需求、设计和文档中的非代码缺陷,这些缺陷不能用旧的“代码行”指标来度量。

另一个重要原因是,所有主要生产力和质量基准数据收集都使用功能点指标,且使用 IFPUG 功能点表示的数据占到所有已知基准数据的 85% 左右。

也不是不能使用其他的度量指标来做数据归一化,但如果使用这些度量指标的结果要与行业基准数据,例如那些由国际软件基准组织(ISBSG)发布的数据作比较,则使用 IFPUG 功能点最方便。在稍后关于缺陷预测的讨论中将会给出使用除 IFPUG 功能点以外的其他度量指标。

有趣的是,将缺陷来源、严重性及缺陷原因等因素进行组合来检查每一个因素的近似发生频率。

表 9-1 显示了开发期间应用软件中这些因素的组合情况。因此,表 9-1 显示了潜在缺陷或者开发期间和发布之后所遭遇到缺陷的可能数目。该表只显示了严重性等级为 1 和 2 的缺陷。

表 9-1 软件潜在缺陷概况

缺陷来源	每功能点缺陷	严重性 1 缺陷	严重性 2 缺陷	最常见缺陷原因
需求	1.00	11.00%	15.00%	遗漏类
设计	1.25	15.00%	20.00%	遗漏类
代码	1.75	70.00%	57.00%	功能类
文档	0.60	1.00%	1.00%	歧义性
不良修复	0.40	3.00%	7.00%	功能类
合计	5.00	100.00%	100.00%	遗漏类

潜在缺陷数据来自于软件缺陷和缺陷去除效率的长期科学研究,这些研究由诸如 IBM 软件质量保证部门这样的组织实施,他们研究软件质量已经超过 35 年了。

其他一些公司也进行缺陷和缺陷去除效率的长期研究，这些公司和组织包括 AT&T、Coverity、Computer Aid Inc.(CAI)、Dovél Technologies、Motorola、软件生产力研究所 (SPR)、Galorath Associates、David Consulting 集团、质量和生产力管理集团 (Quality and Productivity Management Group, QPMG)、Unisys、微软等。

大多数这样的研究都是由大公司而不是大学实施的，因为学术界还没有真正建立起能够执行持续 10 年以上的长期研究措施。

代码漏洞 (bugs) 和代码缺陷 (defects) 都是开发期间数量最多的缺陷，它们也是最容易发现和消除的。正式审查、静态分析以及软件测试相结合可以消灭超过 95% 的代码缺陷，有时甚至高达 99%。需求缺陷和不良修复是最难以消除的缺陷类型。

表 9-2 以表 9-1 为基础，说明了应用软件交付给用户时仍然会出现的潜在缺陷。表 9-2 中显示了大约 2009 年美国软件交付缺陷的近似平均水平。请注意缺陷去除效率上缺陷来源的变化。

表 9-2 已交付软件缺陷概况

缺陷来源	每功能点缺陷	去除效率	每功能点交付缺陷	最常见缺陷原因
需求	1.00	70.00%	0.30	功能类
设计	1.25	85.00%	0.19	功能类
代码	1.75	95.00%	0.09	功能类
文档	0.60	91.00%	0.05	遗漏类
不良修复	0.40	70.00%	0.12	功能类
合计	5.00	85.02%	0.75	功能类

有意思的是，当软件交付给客户时需求缺陷成了最大数量的缺陷，主要是因为它们最难以预防也最难以发现。只有使用正式需求收集和正式需求审查相结合的方法可以改善发现和去除需求缺陷的这种状况。

如果不加以预防和消除，需求缺陷和设计缺陷最终都会进入产品代码。这些都不是代码本身的错误（例如分支到一个错误地址）而是更严重和更深层次的各种错误或缺陷。

本章早些时候已指出，需求缺陷不能单靠测试来发现和消除。如果需求缺陷没有通过审查加以预防和消除，则所有依据软件需求而创建的测试用例将会接受而不会识别出这些需求缺陷。

表 9-2 反映了美国交付软件缺陷的近似平均水平，那些看起来比较粗心的需求收集方法有：瀑布开发模式；CMMI 1 级；没有需求、设计或代码正式审查；没有静态分析；仅使用 5 种形式的测试：（1）单元测试，（2）新功能测试，（3）回归测试，（4）系统测试，（5）验收测试。

还需注意的是，在开发过程中软件需求会以每月 1% 到 2% 的速度继续增长和变化。这些变化中的需求比那些原始需求具有更高的潜在缺陷和更低的缺陷去除效率水平。这也是为什么需求缺陷比任何其他来源的缺陷导致更多问题的另一个原因。

软件需求是软件缺陷最棘手的来源。然而，诸如联合应用设计 (JAD)、质量功能展开 (QFD)、六西格玛分析、根本原因分析、敏捷开发实践中的用户参与、原型分析以及正式需

求审查的使用等方法都可以协助控制需求缺陷。

表 9-3 显示了如果使用缺陷预防和缺陷去除活动最佳组合的话软件质量看上去可能像什么样子。该表假定使用了正式的需求方法和严格的开发方法,例如实践使用的团队软件过程(TSP)或 CMMI 高等级、原型法及 JAD、所有可交付物的正式审查、代码静态分析以及完整的 8 个测试阶段:(1)单元测试,(2)新功能测试,(3)回归测试,(4)性能测试,(5)安全测试,(6)可用性测试,(7)系统测试以及(8)验收测试。

表 9-3 还假设使用了一个软件质量保证(Software Quality Assurance, SQA)团队和严格的软件缺陷报告。该报告涵盖了起始于软件需求阶段,经过正式审查、静态分析和测试等阶段,一直到多年客户缺陷报告、维护和功能增强等领域的数据。表 9-1 到 9-3 中显示的累积数据需要历经多年的长期数据收集。

表 9-3 最优缺陷预防和缺陷去除活动

缺陷来源	每功能点缺陷	去除效率	每功能点交付缺陷	最常见缺陷原因
需求	0.50	95.00%	0.03	遗漏类
设计	0.75	97.00%	0.02	遗漏类
代码	0.50	99.00%	0.01	功能类
文档	0.40	96.00%	0.02	遗漏类
不良修复	0.20	92.00%	0.02	功能类
合计	2.35	96.40%	0.08	遗漏类

这种组合将促使潜在缺陷降低超过 50%,累积缺陷去除效率从目前的平均 85% 提高到 96% 以上。

软件行业的质量水平也有可能比表 9-3 中所显示结果更好,但需要额外的方法,比如提供全套认证的可重用材料。

表 9-2 和表 9-3 所显示的都是现实软件开发结果的过度简化。潜在缺陷因不同的应用规模和其他因素而有所变化。缺陷去除效率水平也随应用规模而变化。不良修复注入同样会随着缺陷来源的不同而不同。潜在缺陷和缺陷去除效率水平都会因开发方法、CMMI 等级以及其他因素而有所不同。这些将在本章稍后关于缺陷预测的章节中进行讨论。

由于软件行业使用了许多不同的质量定义,最好一开始就弄清楚哪些质量因素是可预测和可度量的而哪些不是。为理清各种质量定义的相关性,笔者开发了一个软件质量因素的 10 点评分方法。

- 如果一个因素导致质量改进,它的最高得分是 3 分。
- 如果一个因素导致客户满意度的提高,它的最高得分也是 3 分。
- 如果一个因素导致团队士气提高,它的最高得分是 2 分。
- 如果一个因素是可预测的,其最高得分是 1 分。
- 如果一个因素是可度量的,其最高得分是 1 分。
- 最高总得分是 10 分。
- 最低得分是 0 分。

表 9-4 列出了本章讨论的所有质量因素,以上述评分方法进行排名。表 9-4 显示一个具

体质量因素是否是可度量的和可预测的，以及基于软件客户调查结果得出的该因素与质量的相关度。它还包括一个关于某因素是否能在使用它的组织内提高软件质量的加权判定。

表 9-4 质量因素对质量重要性排名

	可度量属性	可预测属性	质量相关性	评分
最佳质量定义				
潜在缺陷	是	是	很高	10.00
缺陷去除效率	是	是	很高	10.00
缺陷严重性等级	是	是	很高	10.00
缺陷来源	是	是	很高	10.00
可靠性	是	是	很高	10.00
好的质量定义				
“毒性”需求	是	否	很高	9.50
遗漏的需求	是	否	很高	9.50
符合需求	是	否	很高	9.00
过度需求	是	否	中等	9.00
可用性	是	是	很高	8.00
可测试性	是	是	高	8.00
缺陷原因	是	否	很高	8.00
一般质量定义				
可维护性	是	是	高	7.00
可理解性	是	是	中等	6.00
可跟踪性	是	否	低	6.00
可修改性	是	否	中等	5.00
可验证性	是	否	中等	5.00
差的质量定义				
可移植性	是	是	低	4.00
可扩张性	是	否	低	3.00
可伸缩性	是	否	低	2.00
可互操作性	是	否	低	1.00
可生存性	是	否	低	1.00
歧义性	否	否	低	0.00
灵活性	否	否	低	0.00
可管理性	否	否	低	0.00
可操作性	否	否	低	0.00

随着时间推移，得分为 10 分的质量定义在提高软件质量方面已成为最有效的定义。按照惯例，得分高于 7 分的质量定义都是有用的。然而，得分低于 5 分的质量定义根本没有任何可用经验数据证明其可以改进软件质量。

虽然表 9-4 有些主观，但至少它提供了评判软件产业所使用质量因素的相关性和重要性的数学基础，而不再仅仅是一组含混不清、模棱两可的主观质量因素。根本地讲，表 9-4

说明了以下几点:

1. “符合需求”是非常危险的,除非那些不正确、毒性或危险的需求都被清除掉。这个质量定义已超过30年没有表现出任何质量改善了。
2. 绝大多数“特性”质量定义很难度量,而且很多“特性”意义不大。还有些“特性”是无法度量的。没有任何“特性”定义会导致实际软件质量改善。
3. 量化潜在缺陷和缺陷去除效率水平在软件质量改进和客户满意度水平上都有最大影响。

如果软件工程要从一门手艺(craft)或艺术形式(art form)演变为一个真正的工程领域,有必要把软件质量构筑在牢固的定量基础之上,远离模糊不清、主观的质量定义。当然,仍然有需要这些质量定义的地方,但它们不应该成为软件质量的首要定义。

9.2.8 预测软件潜在缺陷

要预测软件质量,有必要先对软件质量进行度量。由于像IBM这样的公司度量软件已经超过40年了,所以最好的可用数据就来自于这些拥有全生命周期软件质量度量计划的公司。这种全生命周期软件质量度量起始于软件需求,经过开发阶段,一直延伸到发布之后客户报告缺陷,只要该软件还在使用,这可能会有25年或者更长时间。另一个最好的数据来源是基准与商业软件评估工具公司,因为这些公司收集各种软件的质量及生产力历史数据。

软件缺陷有5种不同的来源,因此获取软件潜在缺陷有用近似数据的最快方法就是使用IFPUG功能点指标。

使用功能点来预测软件潜在缺陷的基本规则是:软件潜在缺陷的数目是以功能点数目计算的软件应用规模的1.25次幂。这个结果就是软件应用潜在缺陷的合理近似值,这些应用软件的规模可以小到10个功能点,大到5000个功能点。

对于CMMI较高等级、敏捷、RUP及团队软件过程(TSP),这个经验规则的指数可能需要向下调整。但由于该规则意在尽可能早在软件项目中实施,所以在成本真正花出去之前,它仍然提供了一个有用的潜在缺陷粗略估计。读者可能需要尝试使用本地数据找出能够给出本地质量的潜在缺陷结果的合理指数。

表9-5给出了美国软件项目粗略的平均潜在缺陷。回想一下,这里的软件潜在缺陷值是5个缺陷来源的总和:需求缺陷、设计缺陷、代码缺陷、文档缺陷和不良修复注入。

如表9-5所示,潜在缺陷的数量随着应用规模的增大而增大。当然,正如将在稍后“缺陷预防”部分讨论的那样,其他因素也会增加或减少潜在缺陷。

虽然了解总的潜在缺陷对预测软件质量非常有用,但知道软件缺陷在5个来源上的分布也同样受益良多。表9-6用近似平均值说明了典型的缺陷分布百分比。

表 9-5 美国平均软件潜在缺陷

功能点数	每功能点缺陷数	潜在缺陷
1	1.50	2
10	2.34	23
100	3.04	304
1 000	4.62	4 621
10 000	6.16	61 643
100 000	7.77	777 143
1 000 000	8.56	8 557 143
平均	4.86	1 342 983

把表 9-6 中所示分布应用到一个具有 1500 个功能点的示例应用程序上，表 9-7 显示了在开发过程中及由客户发现的近似潜在缺陷总数。

以上整个简单示例并不能取代商业质量评估工具，比如 KnowledgePlan 和 SEER，这些商业工具可以根据某些因素调整它们的预测结果。其中涉及的因素包括 CMMI 等级、开发方法（如敏捷、TSP 或 RUP）、是否使用正式审查、是否使用静态分析以及可以导致潜在缺陷和缺陷去除效率水平发生变化的其他因素。

表 9-6 缺陷分布百分比（来源）

缺陷来源	每功能点缺陷数	占总缺陷的百分比
需求	1.00	20.00
设计	1.25	25.00
源代码	1.75	35.00
用户文档	0.60	12.00
不良修复	0.40	8.00
合计	5.00	100.00

表 9-7 实例应用的潜在缺陷^①

缺陷来源	每功能点缺陷数	潜在缺陷	占总缺陷的百分比
需求	1.00	1500	20.00
设计	1.25	1875	25.00
源代码	1.75	2625	35.00
用户文档	0.60	900	12.00
不良修复	0.40	600	8.00
合计	5.00	7500	100.00

① 应用规模 = 1500 功能点。

经验法则永远不会非常精确，但其方便性和易用性为软件潜在缺陷预测的粗略估计和前期估算提供了重大价值。即便如此，这些规则不应该用于软件合同或需要严格估算结果的地方。

9.2.9 代码缺陷预测

使用功能点度量指标作为软件质量预测的整体工具非常有用，因为现代软件项目中的非代码缺陷总数已在数量上超过了代码缺陷。话虽这么说，代码缺陷仍然比任何其他单个来源的缺陷数量要多一些。

由于以下 6 个原因，预测代码缺陷相当棘手：

1. 软件行业现存有超过 2500 种编程语言，就缺陷来源而言，它们各不相同。
2. 大多数现代应用软件使用超过一门语言，甚至有些应用软件使用了多达 15 门不同的编程语言。
3. 从使用相同编程语言编写相同测试程序的程序员抽样表明，受测程序员在代码量上的变化范围非常大，甚至超过了 10 倍之多。个人技能和编程风格不同导致在解决同样问题时的代码编写量、潜在缺陷和生产力方面差异显著。
4. 代码行数可以使用物理行数也可以使用逻辑语句数来计算。对于某些编程语言，这两种计算方法的计算结果是相同的，但对其他编程语言来说，这两种方法的计算结果差异可能会高达 5 倍。

5. 对于许多编程语言, 如 Visual Basic, 大量编程工作使用按钮和下拉式菜单的方法完成。因此, 编程是在没有涉及程序源代码的情况下完成的。目前, 还没有有效规则来计算这种语言的源代码数量。

6. 来自遗留程序或可重用代码库的源代码重用相当普遍。如果重用代码经过审定, 与新编写的代码相比, 重用代码所包含的缺陷更少。

要预测代码缺陷, 有必要知道编程语言等级。编程语言等级的概念经常以短语形式被非正式地使用, 如“高级语言”或“低级语言”。

在 20 世纪 70 年代的 IBM, 当首次进行代码缺陷预测研究时, 首要任务就是制定一个编程语言等级的正式数学定义。在 IBM 内部, 编程语言等级被定义为: 与高级语言 1 个语句所实现功能等效的基本汇编语言语句数。

按照这个定义, COBOL 是 3 级语言, 因为要实现 1 个 COBOL 语句相同的功能需要 3 个基本汇编语句。同样, SMALLTALK 是 15 级语言。

(在功能点发明之前的很多年里, IBM 一直使用“等价汇编语句”作为评估非代码工作, 如用户手册编写等的基础。这样, 不是把在 PL/S 里编写程序所需工作量的 10% 作为文档工作预算基础, 而是以基本汇编语言语句计算的代码工作量的 10% 为基础来估算其预算。这种方式非常原始但合理有效。)

而正是由于对评估所用等价汇编语句方法的强烈不满, 促使 IBM 任命 Allan Albrecht 及其同事来开发功能点度量指标。

更多编程语言, 如 APL、Forth、Jovial 以及其他一些语言陆续出现, 促使 IBM 需要开发一种度量指标和评估方法, 以更精确的方式处理非编码和编码工作。同时, IBM 希望这种新的度量指标和评估方法还可以预测代码缺陷。

宏汇编语言的使用带来了重用的概念, 但同时也导致了度量上的难题, 即如何计算软件应用中重用代码或任何其他重用材料。这里的解决方案是将生产力和质量分为两个主题: (1) 开发, (2) 交付。

“开发”涉及任何必须从头开始构建的代码和材料。“交付”涉及最终交付的软件应用, 包括重用的材料。例如, 使用宏汇编语言, “开发生产力”的生产率是每月 300 行代码。但由于宏扩展方式的重用代码, “交付生产力”可能高达每月 750 行代码。

同样的区分也会影响质量。假定一个程序中有 1000 行新代码和 100 行重用代码, 那么每千行新代码中可能有 15 个缺陷, 但每千行重用代码则没有缺陷。

这是一个重要的业务区别, 但甚至直到 2009 年软件行业仍不能很好地理解它。软件工程的真正目的是提高交付生产力及其质量, 而非开发生产力及其质量。

大约在 1975 年, 开发出功能点指标方法之后, 人们扩展了“编程语言等级”的定义, 使其包含了“1 个功能点等效的逻辑代码语句数”。例如, COBOL 语言每个功能点在程序和数据分区需要大约 105 个语句。(这种扩展是逆火分析或从源代码到功能点直接转换的数学基础。)

表 9-8 显示了如果使用 15 种不同编程语言编写具有 1000 个功能点的同一程序, 代码规模和代码缺陷的变化情况。该表假定所有语言都恒定具有每千行代码 15 个潜在编码缺

陷。然而，15个语言的语言等级从1到15有所不同，所以1000个功能点产生了截然不同的代码数量。

注意：编程语言的等级是可变的，该变化基于重用代码的数量和对外部功能的调用。表9-8中显示的语言等级仅仅是近似值，且并非固定不变。

表 9-8 15种编程语言每千代码行和功能点缺陷示例^①

等级	抽样语言	每功能点源代码	每千功能点源代码	编码缺陷	每功能点缺陷
1	汇编	320	320 000	4 800	4.80
2	C	160	160 000	2 400	2.40
3	COBOL	107	106 667	1 600	1.60
4	PL/I	80	80 000	1 200	1.20
5	Ada 95	64	64 000	960	0.96
6	Java	53	53 333	800	0.80
7	Ruby	46	45 714	686	0.69
8	E	40	40 000	600	0.60
9	Perl	36	35 556	533	0.53
10	C++	32	32 000	480	0.48
11	C#	29	29 091	436	0.44
12	VB	27	26 667	400	0.40
13	ASP.NET	25	24 615	369	0.37
14	Objective C	23	22 857	343	0.34
15	Smalltalk	21	21 333	320	0.32

① 假定所有语言都具有固定的每千行代码15个缺陷。

由表9-8可见，为预测代码缺陷，关键是要知道将使用的编程语言（可能会使用多种编程语言）以及用功能点和代码行指标度量的应用程序规模。

在同一个应用软件中混合使用2种或多种编程语言的情况有点儿棘手，但具有多语言估算能力的商业软件成本估算工具（如KnowledgePlan）可以很好地处理这种问题。重用代码同时也增加了代码缺陷预测的复杂性。

为了说明相同应用软件中使用多种编程语言情况下的估算结果，让我们考虑两个研究案例。

案例A中有3种不同的编程语言，每种语言都有以逻辑语句计算的1000行代码。案例B中有3种与案例A相同的编程语言，但每种语言包含25个功能点。

案例A中，源代码总数为3000行代码；总功能点数为73个；总潜在代码缺陷为45个。

当案例B的假设条件更改为每种语言固定25个功能点时，功能点总数仅仅从73个变为75个，但是源代码数量几乎翻番，缺陷数目也是一样。这是因为最低级编程语言C语言所受到的影响更大一些。

对案例A和案例B可以很容易看出，预测多语言应用程序的规模大小或质量要远比预测单语言应用程序的规模或质量复杂得多。

案例 A: 3 种语言 (每种语言 1000 行代码)

语言	等级	代码行数	每功能点代码行数	功能点数	潜在缺陷
C	2.00	1000	160	6	15
Java	6.00	1000	53	19	15
SmallTalk	15.00	1000	21	48	15
合计		3000		73	45
平均	7.76		41		

案例 B: 3 种语言 (每种语言 25 个功能点)

语言	等级	代码行数	每功能点代码行数	功能点数	潜在缺陷
C	2.00	4000	160	25	60
Java	6.00	1325	53	25	20
SmallTalk	15.00	525	21	25	8
合计		5850		75	88
平均	4.10		78		

有意思的是,把案例 A 和案例 B 并排放在一起以突显其差异。注意,案例 B 中最低级编程语言 C 语言的影响,它既增加了代码量,也提高了潜在缺陷数量。

案例 A 和案例 B 都过度简化了实际软件开发项目中的问题,每个案例都使用了常量的数据项,而真实项目中,它们是变化的。例如,每千行代码 15 个缺陷的固定代码缺陷数实际上是可变的,其变化从每千行代码少于 5 个到超过 25 个缺陷不等。

每个功能点的源代码语句数目也是变化的,而不同编程语言的源代码语句数据也有所不同,变化范围为围绕其所标称语言“等级”源代码量的平均值上下浮动 1 到 2 个语句。

这些变化解释了质量和缺陷水平预测为什么严重依赖于质量和缺陷水平度量。上述例子也说明了软件质量定义为什么需要既可度量又可预测。

其他一些变化也能影响预测软件规模及缺陷的能力。例如,假设案例 A 中重用代码占代码总量的 50%,还假定重用代码经过测试认证且具有零缺陷。现在,缺陷预测计算需要包含这些重用代码。在本例子中,重用代码可以降低潜在缺陷 50%。

当应用程序规模用于生产力计算时,很有必要确定,到底是开发生产力还是交付生产力或者两者都是我们真正感兴趣的数据。

软件缺陷预测可以达到相当不错的精确度,但其计算却不是轻而易举的事情,它们需要包含大量只有经过仔细测量才能确定数值的变量。

源代码 (逻辑语句)	案例 A	案例 B
C	1000	4 000
Java	1000	1 325
Smalltalk	1000	525
总代码行数	3000	5 850
总千行代码	3.00	5.85
功能点数	73	75
代码缺陷	45	88
每千行代码缺陷数	15	15
每功能点缺陷数	0.62	1.17

9.2.10 需求蔓延对软件质量的影响

分别在需求阶段结束时和应用交付时所做的功能点分析表明，在设计和编码阶段，需求以每月超过 1% 的速率进行增长和变更。需求蔓延的总增长范围，从少于总需求的 10% 到超过总需求的 50% 不等。（曾有个项目的需求增长超过了 200%。）

举例说明，如果一个应用软件在初始需求阶段结束时的规模是 1000 个功能点，那么每月将会因为增加新需求而添加至少 10 个新功能点。这种情况可能会持续 6 个月，所以该应用软件的规模将会从 1000 个功能点增长到 1060 个。对于小型软件项目，需求蔓延不是严重问题，充其量会带来诸多不便。

大型软件应用开发周期较长，通常其需求变更的比率也较高。对于一个初始规模为 10 000 个功能点的应用，其新需求增长率可能达到每月 125 个功能点，持续 20 个月。最终交付的软件应用规模可能达到 12 500 个功能点，而不是初始的 10 000 个。这个示例表明，需求增加达到 25% 将对软件开发进度、成本、质量以及交付缺陷产生重大影响。

由于新需求及需求变更发生在初始需求阶段之后的开发阶段，通常处理比较仓促。结果是，新增需求及变更需求的潜在缺陷要比初始需求高出大约 10%。毒性需求和设计错误同样如此。基于软件工程团队的进度压力，代码缺陷可能会也可能不会有所增长。

需求蔓延往往能够规避正式审查，为其创建的测试用例数量也很少。结果，缺陷去除效率比毒性需求和设计错误的去除效率至少 5%。这个估计结果对代码错误似乎也同样成立，除非应用软件开发使用 C 语言或者 Java 语言编写。由 C 或 Java 语言编码的应用软件可以使用静态分析工具，在代码错误上能够达到较高水平的缺陷去除效率。

需求蔓延的高潜在缺陷和低缺陷去除效率共同作用，导致源自需求变更的交付缺陷率远大于任何其他错误来源。这已经成为软件行业长期存在的顽疾。

需求蔓延和低于最佳水平的缺陷预防及缺陷去除实践是软件项目取消、工期延误及成本超支的主要原因。

正如稍后在缺陷预防和缺陷去除部分将会讨论的，目前已经有许多技术可以有效地将需求蔓延造成的危害降低到最小限度。然而，这些有效方法，例如需求和设计的正式审查，同样并未在软件行业广泛应用。

9.3 软件质量度量

尽管缺陷去除效率对于软件项目的成功至关重要，但通常很少有人度量缺陷去除效率或软件质量。从走访的美国、欧洲和亚洲的 300 余家公司看，笔者发现如下各种质量度量情况的分布概率。

度量软件缺陷去除效率的数学方法并不复杂。在数据收集和计算过程中，需要如下 12 步操作来量化缺陷去除效率水平：

1. 从需求阶段开始，对每一个

根本没有任何质量度量	44%
只度量客户报告的缺陷	30%
度量测试活动和客户报告的缺陷	18%
度量正式审查、静态分析、测试活动和客户报告的缺陷	7%
使用志愿者来度量个人缺陷去除	1%
整体分布	100%

出现的缺陷，记录以积累数据。

2. 当修复缺陷时，为每一个报告的缺陷分配严重性级别。
3. 度量每个缺陷去除活动去除了多少缺陷。
4. 使用根本原因分析来识别高严重性缺陷的来源。
5. 同时也度量无效缺陷、重复缺陷及误报。
6. 软件发布之后，度量客户报告的缺陷。
7. 记录缺陷预防、缺陷去除和缺陷修复的工作小时数。
8. 选定一个固定的时间点，如发布之后 90 天来计算缺陷去除效率。
9. 使用志愿者来记录个人缺陷去除活动（例如桌面检查）数据。
10. 为整个软件开发周期各系列阶段计算累积的缺陷去除效率。
11. 为整个软件开发周期各系列阶段的每个步骤计算缺陷去除效率。
12. 使用上述所得数据来改进缺陷预防和缺陷去除措施。

与这些信息的价值相比，度量缺陷去除效率水平所需要付出的努力和成本微不足道。度量每一个缺陷及其相关的修复工作量所需要的总工作量只有大约一小时。这些时间中，花费在客户报告缺陷和内部报告缺陷上的时间基本各占一半。

然而，根据开发团队如何处理需求和设计的方法不同，第 4 步的根本原因分析可能需要花费一些额外时间。

缺陷去除效率度量的价值包括以下几点：

- ☐ 发现和修复缺陷是任何软件开发中最昂贵的活动，因而降低这些成本将带来巨大的投资回报。
- ☐ 过多缺陷是造成软件项目进度延误的主要原因，因而在所有可交付物中降低缺陷将缩短开发周期。
- ☐ 交付缺陷是发布之后头两年软件维护的主要工作，故提高缺陷去除效率可以降低软件维护成本。
- ☐ 客户满意度与交付缺陷数目成反比关系，故降低交付缺陷数目将直接取悦客户。
- ☐ 团队士气与有效的缺陷预防和缺陷去除水平成正比关系。

在稍后的软件质量经济价值部分，将把这些好处进行量化以说明缺陷预防和缺陷去除的整体价值。

许多公司和政府组织会跟踪在静态分析、软件测试中发现的软件缺陷，以及客户所报告的缺陷。实际上，目前已经有许多可用的商业软件缺陷跟踪工具。

这些工具通常跟踪缺陷的症状、包含缺陷的应用、硬件和软件平台，以及其他种类的指示性数据，如发布版本号、内部构建号等。

然而，更多成熟组织还使用了需求、设计的正式审查以及其他材料。这些公司经常使用静态分析和软件测试，因而度量了比那些由普通测试发现的代码缺陷更广范围的缺陷。

为了在根本原因分析和其他形式的缺陷预防活动中使用扩展了的缺陷数据，需要一些额外信息。这些额外主题包括：

缺陷发现点 记录任何特定缺陷发现点的信息非常重要。既然无法正常地通过测试发

现需求缺陷，那么尝试识别非代码缺陷发现点就同等重要。

总的来说，需求和设计中的非代码缺陷要比编码缺陷多得多，而且严重性更高。非代码缺陷的缺陷修复成本经常比编码缺陷的修复成本高。注意，软件行业目前共有超过 17 种软件测试方法，而且不同公司所使用的各个测试阶段的名称不尽相同。

缺陷发现日期：_____

缺陷发现点：

- ☐ 客户缺陷报告
- ☐ 质量保证缺陷报告
- ☐ 测试阶段_____缺陷报告
- ☐ 静态分析缺陷报告
- ☐ 代码审查缺陷报告
- ☐ 文档审查缺陷报告
- ☐ 设计审查缺陷报告
- ☐ 架构审查缺陷报告
- ☐ 需求审查缺陷报告
- ☐ 其他_____缺陷报告

缺陷来源点 记录软件缺陷来源信息也非常重要。这个信息需要在不同团队之间仔细分析，因而很多公司将缺陷来源研究局限于高严重性的缺陷，比如严重性 1 或者严重性 2 的缺陷。

缺陷来源日期：_____

缺陷来源点：

- ☐ 应用软件名称
- ☐ 发布版本号
- ☐ 内部构建号
- ☐ 源代码（内部）
- ☐ 源代码（遗留应用的重用代码）
- ☐ 源代码（商业来源的重用代码）
- ☐ 源代码（商业软件包）
- ☐ 源代码（不良修复或以前的缺陷修复代码）
- ☐ 用户手册
- ☐ 设计文档
- ☐ 架构文档
- ☐ 需求文档
- ☐ 其他_____来源点

理想情况下，出现缺陷和发现缺陷之间的时间间隔一般不会超过一个月，理想情况下不超过一周。在一个项目阶段（如需求和设计阶段）中出现的缺陷应该在同一个项目阶段中发现并获得修复，这一点非常重要。

当出现缺陷和发现缺陷的时间间隔较长时,比如一个设计问题直到系统测试阶段才发现,这就明显预示着软件开发和质量控制过程需要改进。

缩短缺陷出现和缺陷发现之间时间间隔的最好解决方案是需求、设计和其他可交付物的正式审查。静态分析和代码审查也都对缩短缺陷出现和缺陷发现之间时间间隔具有重要帮助。

表 9-9 展示了用于各种缺陷来源的缺陷发现点的最佳案例。

在找出那些微妙而复杂的软件缺陷和问题方面,正式审查通常表现最佳。由于有时并未为它们创建测试用例,这些软件缺陷和问题通常很难通过软件测试而发现。千年虫问题的例子就说明了一个本可以通过测试发现的软件问题,只是 2 位数字的日期被错误地认为是可接受的,因而实际上测试未能发现该问题。代码审查对发现一些比较“狡猾”的问题(例如可能逃过测试甚至静态分析的安全漏洞)也非常有用。

在发现常见编码错误上,静态分析是最优秀的方法。这些编码错误包括跳转到不正确位置、溢出条件、拙劣的错误处理等。测试之前的静态分析作为测试的一种辅助手段,可以降低测试成本。

在发现那些只在代码运行时才会出现的软件问题方面,软件测试是最好方法。比如软件性能问题、可用性问题、接口问题,以及其他如屏幕显示及报告的数学错误和格式错误等。

鉴于软件错误和缺陷的多样性特点,很明显,所有这三种缺陷去除方法对于软件成功都很重要:正式审查、静态分析和软件测试。

表 9-10 表明了这样一个事实:缺陷出现和缺陷发现点之间的长时间延误会导致非常令人担忧的后果。长时间延迟也会导致不良修复注入,在试图修复旧缺陷时意外引入新的缺陷。

在最坏情况下,需求缺陷直到软件部署时才发现,而设计缺陷直到系统测试时才发现。这个时候,在不延长项目整体进度计划的情况下修复这些缺陷将非常困难。需要注意的是,最坏情况下,测试用例自身的缺陷和错误从未被发现,因而这些测试用例缺陷使得很多软件发布变得更加糟糕。

缺陷预防和尽早缺陷去除要远比单单依赖软件测试更具有成本效益。

其他质量度量包括以下各项中的某些或全部:

质量挣值 (EQV) 既然预测潜在缺陷和缺陷去除效率水平可行,诸如 IBM 等公司已经使用一种形式的“挣值”方法,将审查、静态分析和软件测试等活动所发现缺陷的预测结果和实际发现的缺陷数目进行对比,还将预测的和实际的缺陷去除成本进行对比。

如果发现的缺陷比预测的少,则可以用根本原因分析来查明软件质量是否真的比计划的好,还是因为缺陷去除做得不够好。(当这一切发生时,通常质量比较好。)

表 9-9 缺陷发现点的最佳案例

缺陷来源	最优的缺陷发现案例
需求	需求审查
设计	设计审查
编码	静态分析
不良修复	静态分析
文档	编辑
测试用例	测试用例审查

表 9-10 缺陷发现点的糟糕案例

缺陷来源	最晚的缺陷发现点
需求	部署
设计	系统测试
编码	新功能测试
不良修复	回归测试
文档	部署
测试用例	没有发现

如果发现的缺陷比预测的多，则可以用根本原因分析来查明软件质量是否真的比计划的差，还是因为缺陷去除比预期的更有效。（通常此时软件质量比较糟糕。）

质量成本（COQ） 总体而言，发现和修复缺陷的费用是软件行业历史上已知最昂贵的活动成本。因此，以计算质量成本的方式收集所付出工作和成本数据非常重要。

但是，对于软件，需要对正常质量成本（COQ）计算进行调整，以符合软件工程的具体实际。一般地，数据以小时数形式记录，然后转换成应付工资、负债率以及其他项目应付支出项的成本。

- ☐ 缺陷发现活动_____
- ☐ 缺陷预防活动_____
- ☐ 用户报告的缺陷工作量
- ☐ 用户报告的缺陷损害
- ☐ 审查准备小时数
- ☐ 静态分析准备小时数
- ☐ 测试准备小时数
- ☐ 缺陷发现小时数
- ☐ 缺陷报告小时数
- ☐ 缺陷分析小时数
- ☐ 缺陷修复小时数
- ☐ 缺陷审查小时数
- ☐ 缺陷静态分析小时数
- ☐ 用于缺陷的测试阶段
- ☐ 为缺陷创建的测试用例
- ☐ 缺陷测试小时数

在没有实际分析“平均缺陷成本”指标如何起作用的情况下，软件行业已经一直在使用该度量指标了。实际上，数以百计的文章和书籍鹦鹉学舌地使用了类似诸如“软件发布之后的缺陷修复成本是编码阶段的100倍”这样的措辞，或者类似这样的表述。这些教条表述的要点是平均缺陷成本随着缺陷发现的延后而稳步上升。

很少有人能认识到，发现缺陷最多时平均缺陷成本总是最便宜，而发现缺陷最少时平均缺陷成本最昂贵。实际上，就像通常计算的，这个指标违反了标准的经济学假设，因为它忽视了固定成本。平均缺陷成本实际上对质量改进不利，它会造成漏洞百出的软件应用这种最差结果。

以下是对平均缺陷成本为什么对质量不利并造成了漏洞百出的软件应用这种最坏结果的分析。相同的数学分析也同样表明早发现缺陷为什么比晚发现似乎更加便宜。

此外，即使应用软件实现了零缺陷，仍然有大量的评估和测试活动需要支付费用。很明显，平均缺陷成本指标对零缺陷应用没有任何实际用处。

由于平均缺陷成本通常的度量方式，因此随着软件质量提高，平均缺陷成本稳步升高，直到软件达到零缺陷。此时，平均缺陷成本指标根本不能使用。

正如千代码行 (KLOC) 指标的错误一样, 平均缺陷成本指标的主要错误来源是它忽略了固定成本。下述 3 个例子将说明, 随着软件质量的提高平均缺陷成本表现如何。

所有 3 个例子中, 案例 A、案例 B 和案例 C, 假设测试人员每周工作 40 小时, 劳动报酬是每周 2500 美元或者每小时 62.50 美元。假设所有 3 个要测试的软件功能的规模都是 100 个功能点。

案例 A: 质量低劣

假设测试员花费 15 小时编写测试用例, 10 小时运行测试用例, 15 小时修复 10 个缺陷。花费总时间是 40 小时, 总成本是 2500 美元。由于发现 10 个缺陷, 每个缺陷的成本是 250 美元。进行测试的这一星期中, 每个功能点的成本是 25.00 美元。

案例 B: 质量优良

在第 2 个案例中, 假定测试员花费 15 小时编写测试用例, 10 小时运行测试用例, 5 小时修复一个缺陷, 这也是唯一一个发现的缺陷。但是, 由于没有其他任务等着要做, 而测试员仍然工作完整一周, 项目仍需支付 40 小时的报酬。

那么本周的总成本仍然是 2500 美元, 所以平均缺陷成本一下子就跳到了 2500 美元。如果刨除 10 小时的闲暇时间, 实际测试和缺陷修复只剩 30 小时, 平均缺陷成本将是 1875 美元。随着质量提高, 平均缺陷成本急剧上升。

现在让我们考虑一下每个功能点的成本。去除闲暇时间, 每个功能点的成本将是 18.75 美元。由此可以轻松看出, 平均缺陷成本随着质量的提高而增长, 从而违反了标准经济度量假设。

然而, 正如我们所看到的, 随着质量的提高, 每个功能点的测试成本降低了。这正好符合标准经济假设。10 个小时的空闲时间说明了另一个问题: 当质量得以改善时, 缺陷减少要快于人员的再分配。

案例 C: 零缺陷应用

在第 3 个案例中, 仍然假设测试员花费 15 个小时编写测试用例, 10 个小时运行测试用例。没有发现任何错误或缺陷。因为没有发现任何缺陷, 平均缺陷成本度量指标根本就不适用。

但是 25 小时的实际工作消耗在了编写和运行测试用例上。如果测试员没有其他任务等待执行, 则测试员仍要每周工作 40 小时, 因而成本是 2500 美元。去除 15 小时的空闲时间, 剩下 25 小时用于实际测试工作, 成本就是 1562 美元。

去掉空闲时间, 每个功能点的成本是 15.63 美元。由此可见, 每个功能点的测试成本随着质量的改进而下降。这里, 每个功能点成本的下降又一次符合了标准的经济学假设。

缺陷修复时间和活动的研究结果也不支持那个“软件发布之后修复一个缺陷的花费是发布之前 100 倍”的格言。典型地, 修复缺陷需要的时间在 15 分钟到 4 小时之间。

有些缺陷的成本非常昂贵。这些缺陷被 IBM 称为“待定缺陷”(abeyant defect)。这些“待定缺陷”通常是由客户报告的, 由于客户环境中某些特殊的软硬件组合, 修复中心无法予以重现。这类缺陷占所有客户报告缺陷的比例不到 5%。

由于缺陷去除活动的固定成本，因而平均缺陷成本总是随着缺陷数量的下降而上升。因为在测试阶段开始时发现的缺陷要比软件发布之后发现的缺陷多得多，这就解释了平均缺陷成本为什么会在软件开发周期末期上升。这是因为编写测试用例、运行测试用例以及软件维护的成本充当了固定成本。

在任何具有高百分比固定成本的制造周期中，单位成本都会随着单位数量的相应减少而上升。制造业经济学的基本事实之一说明了平均缺陷成本为什么有害，且对软件应用的经济分析完全无效。

更为有效的举措是记录花费在各种缺陷去除活动上的小时数。在记录了该时间数据之后，该时间数据可转换为成本数据，而后将小时数转换为标准单位（如每个功能点所需测试工作小时数）而获得归一化（normalized）。

表 9-11^① 显示了上述各种活动小时数数据的一个示例，这些数据在质量成本评估、经济研究以及效益研究中非常有用。该表中的数据表示的是 1 名软件人员在 10 个功能点或者 500 个 Java 语句上各种活动的准备时间。

表 9-11 软件缺陷去除工作付出积累

去除阶段	缺陷去除工作（工作小时数）			
	准备小时数	执行小时数	修复小时数	总小时数
审查：				
需求	0.25	0.25	0.50	1.00
架构	0.25	0.25	0.75	1.25
设计	0.50	0.50	1.00	2.00
源代码	0.75	0.50	0.75	2.00
文档	0.10	0.25	0.25	0.60
静态分析	0.10	0.10	0.25	0.45
测试阶段				
单元测试	0.50	0.20	0.50	1.20
新功能测试	0.50	0.25	0.50	1.25
回归测试	0.25	0.50	1.00	1.75
性能测试	0.50	0.75	1.00	2.25
可用性测试	0.75	0.75	1.00	2.50
安全测试	0.75	1.00	1.50	3.25
系统测试	0.50	0.50	2.00	3.00
独立测试	0.25	0.25	0.75	1.25
Beta 测试	0.10	0.30	1.00	1.40
验收测试	0.10	0.50	1.00	1.60
供应链测试	0.25	0.50	1.25	2.00

① 原书中表 9-11 和表 9-12 没有数据，只显示要收集数据的类型和格式。经作者再三考虑，添加数据举例说明比较合适。故译文版添加了示例数据，所有数据均由作者提供。——译者注

(续)

去除阶段	缺陷去除工作(工作小时数)			总小时数
	准备小时数	执行小时数	修复小时数	
维护:				
客户	0.25	0.00	0.75	1.00
内部 SQA	0.10	0.50	0.75	1.35

当然,知道缺陷去除时间暗含着也收集了缺陷数量和缺陷严重性等级等数据。表 9-12 使用了与表 9-11 相同的活动集合,但是显示的是缺陷数据。这里仍然假设 10 个功能点或者 500 个 Java 语句的规模。表 9-11 和表 9-12 都可以合并到一个大的电子表格中。然而,缺陷数和为缺陷付出的累积工作量往来自不同来源,故可能无法同时获得相应数据。

表 9-12 软件缺陷严重性等级累积

去除阶段	缺陷严重性等级				总缺陷
	严重性 1(重大)	严重性 2(严重)	严重性 3(轻微)	严重性 4(细微)	
审查:					
需求	0	1	1	0	2
架构	0	0	1	0	1
设计	1	0	1	0	2
源代码	0	1	2	0	3
文档	0	0	0	3	3
静态分析	0	1	3	3	7
测试阶段:					
单元测试	0	0	1	0	1
新功能测试	0	0	1	1	2
回归测试	0	0	0	1	1
性能测试	0	1	0	0	1
可用性测试	0	1	0	1	2
安全测试	1	2	0	0	3
系统测试	0	0	2	1	3
独立测试	0	0	1	2	3
Beta 测试	0	1	2	2	5
验收测试	0	0	1	1	2
供应链测试	0	1	0	0	1
维护:					
客户	0	1	2	2	5
内部 SQA	0	0	1	2	3
总缺陷数	2	10	19	19	50

为缺陷而付出的所有努力和已发现缺陷数目都是长期质量改进的重要数据元素。实际上,没有这些数据,质量改进的效果很可能微乎其微,甚至根本就不会有任何改进。

未能记录缺陷数量和为缺陷而付出的工作量是软件工程领域长期以来的不足之处。但是，某些软件开发方法，比如团队软件过程（TSP）和 Rational 统一过程（RUP），确实包括了详细的缺陷度量措施。另一方面，敏捷方法在软件质量度量方面既不强也不始终如一。

为使软件工程成为真正的工程行业而不再仅仅是一门手艺（就像它在 2009 年时的那样），缺陷度量措施、缺陷预测、缺陷预防和缺陷去除需要成为软件工程的重中之重。

度量软件缺陷去除效率

展示和改进软件质量最有效的度量指标之一是“缺陷去除效率”。该指标概念简单但却难以实际应用。其基本想法是，计算借助软件缺陷去除活动（如审查、静态分析和软件测试）发现并修复的软件缺陷占全部软件缺陷的百分比。

使缺陷去除效率计算变得棘手的是软件应用中的缺陷既包括需求、设计以及其他书面可交付物中发现的非代码缺陷，也包括编码缺陷。

表 9-13 举例说明了由需求审查开始到验收测试终止的一整套缺陷去除活动的缺陷去除效率水平。

表 9-13 软件缺陷去除效率水平^①

应用规模（功能点）	1000			
语言	C			
代码规模	125 000			
非代码缺陷	3000			
代码缺陷	2000			
总缺陷数	5000			
去除阶段	不同去除阶段的缺陷去除效率			
	非代码缺陷	代码缺陷	总缺陷	去除效率
审查：				
需求	750	0	750	
架构	200	0	200	
设计	1250	0	1250	
源代码	100	800	900	
文档	250	0	250	
小计	2250	800	3350	67.00%
静态分析	0	800	800	66.67%
测试阶段：				
单元测试	0	50	50	
新功能测试	50	100	150	
回归测试	0	25	25	
性能测试	0	10	10	
可用性测试	50	0	50	
安全性测试	0	20	20	
系统测试	25	50	75	

(续)

去除阶段	不同去除阶段的缺陷去除效率			去除效率
	非代码缺陷	代码缺陷	总缺陷	
独立测试	0	5	5	
Beta 测试	25	15	40	
验收测试	25	15	40	
供应链测试	25	10	35	
小计	200	300	500	58.82%
发布前缺陷	2750	1900	4650	93.00%
维护:				
客户 (90 天)	250	100	350	100.00%
总计	3000	2000	5000	
去除效率	91.67%	95.00%	93.00%	

①假设审查、静态分析和正常测试。

表 9-13 做了许多简化假设。其中,假设之一是所有交付缺陷都将由客户在发布之后的最初 90 天使用期内发现。当然,在实际软件项目中,交付软件中的潜伏缺陷可能在软件中隐藏数月甚至数年而不被发现。但是,软件发布 90 天之后,通常会有新版本的软件发布出来,这使得度量之前发布软件的缺陷变得更加困难。

我们可以看看在一系列不复杂、既不包括正式审查又不包括静态分析的缺陷去除步骤的情况下,会出现什么样的缺陷去除效率水平。

既然来源于需求和设计的非代码缺陷最终都会以自己的方式进入程序代码,如表 9-14 所示,那么在没有任何前期审查和静态分析的情况下,软件测试自身的整体缺陷去除效率水平严重降低。

对比表 9-13 和表 9-14 可见,在发现和去除来源于软件源代码之外的软件缺陷上,一整套的软件缺陷去除活动要比仅仅使用测试这一种方式更加高效而适用。实际上,正式审查和静态分析在发现代码缺陷上也非常胜任,它们还具有提高测试效率、降低测试成本的附加特性。

表 9-14 软件缺陷去除效率水平^①

应用规模 (功能点)	1000			
语言	C			
代码规模	125 000			
非代码缺陷	3000			
代码缺陷	2000			
总缺陷数	5000			
去除阶段	不同去除阶段的缺陷去除效率			去除效率
	非代码缺陷	代码缺陷	总缺陷	
审查:				
需求	0	0	0	
架构	0	0	0	

(续)

去除阶段	不同去除阶段的缺陷去除效率			去除效率
	非代码缺陷	代码缺陷	总缺陷	
设计	0	0	0	
源代码	0	0	0	
文档	0	0	0	
小计	0	0	0	0.00%
静态分析	0	0	0	0.00%
测试阶段：				
单元测试	200	350	550	
新功能测试	450	600	1050	
回归测试	0	100	100	
性能测试	0	50	50	
可用性测试	200	75	275	
安全性测试	0	50	50	
系统测试	300	200	500	
独立测试	50	10	60	
Beta 测试	150	25	175	
验收测试	175	20	195	
供应链测试	75	20	95	
小计	1600	1500	3100	62.00%
发布前缺陷	1600	1500	3100	62.00%
维护：				
客户（90 天）	1400	500	1900	100.00%
总计	3000	2000	5000	
去除效率	53.33%	75.00%	62.00%	

① 假设只有常规测试，没有正式审查和静态分析。

在没有预测试审查和静态分析的情况下，软件测试将会发现数以百计的缺陷。但全部软件测试活动的整体缺陷去除效率反而比包含了正式审查和静态分析的全套缺陷去除活动集合要低。

除了可以提升软件缺陷去除效率水平，将正式审查和静态分析添加到缺陷去除活动集合还可以降低开发和维护成本，缩短开发进度，因为传统上冗长的测试周期常常是软件开发周期的主要部分。事实上，由于软件无法很好地工作以满足发布要求，低劣质量会显著拉长测试进程。

表 9-15 一对一对比了本部分讨论的两个案例的成本结构。案例 X 来自于表 9-13，使用了一个正式审查、静态分析和常规测试的成熟组合。

案例 Y 来自于表 9-14，仅仅使用了常规软件测试活动，没有执行任何正式审查和静态分析活动。

表 9-15 中的成本假设了一个完全承担的每月 10 000 美元的酬劳结构。缺陷去除成本假设包括测试准备、执行及修复所有已发现和识别缺陷的费用。

表 9-15 软件缺陷去除效率成本对比

(案例 X = 审查、静态分析、常规测试)			
(案例 Y = 只有常规测试)			
应用规模 (功能点)	1000		
语言	C		
代码规模	12 5000		
非代码缺陷	3000		
代码缺陷	2000		
总缺陷	5000		
去除阶段	以活动分类的缺陷去除成本		
	案例 X (美元)	案例 Y (美元)	差异
审查:			
需求			
架构			
设计			
源代码			
文档			
小计	168 750 美元	0 美元	168 750 美元
静态分析	81 250 美元	0 美元	81 250 美元
测试阶段:			
单元测试			
新功能测试			
回归测试			
性能测试			
可用性测试			
安全性测试			
系统测试			
独立测试			
Beta 测试			
验收测试			
供应链			
小计	150 000 美元	775 000 美元	-625 000 美元
发布前缺陷	400 000 美元	775 000 美元	-375 000 美元
维护:			
客户 (90 天)	175 000 美元	950 000 美元	-775 000 美元
总成本	575 000 美元	1 725 000 美元	-1 150 000 美元
平均缺陷成本	115.00 美元	345.00 美元	-230.00 美元
平均功能点成本	575.00 美元	1 725.00 美元	-1 150.00 美元
平均代码行成本	4.60 美元	13.80 美元	-9.20 美元
审查、静态分析的 ROI			3.00 美元
开发周期 (自然月)	12.00	16.00	-4.00

除了成本优势之外，卓越的质量控制还与客户满意度和可靠性相互关联。而可靠性和客户满意度与交付缺陷水平成反比关系。

交付软件中缺陷越多，客户越不高兴。此外，平均无故障时间（mean time to failure, MTTF）随着交付缺陷数的下降而增长。这种可靠性相互关系是基于严重性 1 和严重性 2 两类高严重性缺陷的。

表 9-16 显示了软件应用的交付缺陷、以平均无故障（MTTF）小时数形式表示的可靠性和客户满意度之间的近似关系。

表 9-16 交付缺陷、可靠性、客户满意度^①

每 KLOC 交付缺陷 ^②	每功能点缺陷 ^③	平均无故障时间 (MTTF)	客户满意度
0.00	0.00	无限	优秀
1.00	0.13	303	很好
2.00	0.25	223	好
3.00	0.38	157	一般
4.00	0.50	105	差
5.00	0.63	66	很差
6.00	0.75	37	很差
7.00	0.88	17	很差
8.00	1.00	6	提起诉讼
9.00	1.13	1	提起诉讼
10.00	1.25	0	失职

① 假定语言为 C 语言。

② 假定每功能点 125 行代码。

③ 假定严重性 1 和严重性 2 的交付缺陷。

表 9-16 使用的是整数值，所以在这些离散值之间进行插值很有必要。同时，可靠性等级只是大概结果。表 9-13 只讨论了 C 语言，对于其他近 700 门现存语言，在平均功能点缺陷上需要进行调整。在可靠性、客户满意度及它们与交付缺陷水平之间的关系等主题上，还需要更多研究。

然而，过多交付缺陷不仅对客户满意度产生了负面影响，也会导致很多对外包承包商和商业软件开发商的法律诉讼。实际上，在一个软件企业主要股东提起的诉讼案件中，该软件企业股东声称过高交付缺陷水平降低了他们股票的价值。

更好的质量控制是软件工程成功的关键。为使软件工程成为一个真正的工程行业，软件质量需要变得可定义、可预测、可度量 and 可改进。

9.4 软件缺陷预防

短语“缺陷预防”指那些从根本上降低某些类型缺陷发生可能性的方法和技术。缺陷预防的相关文献非常稀缺，而相关学术研究则更为罕见。出现这种情况的原因是缺陷预防研究极其困难，即便是目前最佳的研究结果也有些含混不清、充满歧义。

缺陷预防类似于某些严重疾病(如肺炎或流感)的疫苗接种。统计数据表明,接种疫苗可以降低该人群中该种疾病的发病概率。但是,没有任何证据表明任何患上某种疾病的患者是否接种了该疾病的疫苗。同样,少数已经接种了疫苗的人也可能患上该病,因为疫苗并非100%有效。此外,一些疫苗可能还有严重的、意想不到的副作用。

所有这些问题都同样可能发生在软件缺陷预防方面。尽管有统计数据表明,某些方法,如原型法、联合应用设计(JAD)、质量功能展开(QFD)及参与审查,可以防止发生某些类型的缺陷,但很难证明在没有这些预防性方法的情况下这些类型的缺陷将一定会出现。

缺陷预防的研究方法是,开发功能相似或相同应用软件的两个版本,一个版本使用特别的缺陷预防方法而另一个则不使用该方法。显然,这种研究必须是小规模的。

最容易的缺陷预防实验涉及需求、设计和代码的正式审查。因为审查会记录所有缺陷,那些使用正式审查方法的公司很快就会积累到足够多的数据来分析其缺陷预防措施和缺陷去除活动。

正式审查在缺陷预防方面特别有效,它每年可以降低潜在缺陷超过25%。实际上,正式审查的一个“问题”是,持续使用正式审查方法3年以后,由于很少能再发现更多的缺陷,这使得审查活动也变得无聊起来。

缺陷预防更常用的方式是检查应用程序的大量抽样结果,并关注它们之间在潜在缺陷上的差异。换句话说,如果对比100个使用了原型法的应用程序和100个没有使用原型法的类似应用,那些使用了原型法的应用软件中需求缺陷数量更少吗?原型法的应用软件中需求蔓延更少吗?

这种研究方法只能在那些相当成熟的大型公司内部进行,也就是说,像IBM、AT&T、微软、Raytheon、Lockheed等这样的公司,它们具有非常成熟的软件缺陷和质量度量流程。(为同一个行业的很多公司服务的咨询师经常可以观察到不同公司类似应用软件中缺陷预防的效果。)

然而,基准研究组织有时也会发布这类大规模的统计研究结果。这些组织包括国际软件基准组织(ISBSG)、戴维咨询集团(The David Consulting Group)、软件生产力研究所(SPR)及质量与生产力管理集团(QPMG)等。

此外,曾在软件诉讼案件中担任专家证人的咨询师(比如笔者)可能有机会接触到一些其他渠道无法获得的数据。这些数据说明了那些未能在软件项目中使用缺陷预防措施而导致最终闹上法庭所造成的负面影响。

表9-17列举了已观察到的30种预防软件缺陷出现的方法和技术。尽管该表显示了缺陷预防效率的一个具体百分数,但由于实际数据太稀缺,因而还没有大量实际项目数据能够完全精确地支持这些结果。这些百分比只是一个粗略估计值,只用于表示这些方法在缺陷预防有效性方面的通常排序。

需要注意的是,由于缺陷预防讨论的是减少潜在缺陷,对那些可以减少缺陷的方法,其百分比显示为负值,而正值则表明该方法会增加潜在缺陷。

排名靠前的两项值得讨论。短语“重用自认证来源”表明了正规可重用性,其规格说明书、源代码、测试用例等均已经通过了严格的正式审查和测试阶段,且已经在现场试验

中证明了其可靠性。经过认证的可重用组件可能几乎是零缺陷的，且在任何情况下包含非常少的软件缺陷。对比而言，重用那些未经认证的材料则比较危险。

表 9-17 预防缺陷的方法和技术

预防软件缺陷的活动		缺陷预防效率	预防软件缺陷的活动		缺陷预防效率
1	重用（认证来源）	-80.00%	17	CMM 3	-23.00%
2	参与正式审查	-60.00%	18	Scrum 会议（每日）	-20.00%
3	功能原型	-55.00%	19	代码复杂度分析	-19.00%
4	PSP/TSP	-53.00%	20	用例	-18.00%
5	软件六西格玛	-53.00%	21	重用（未认证来源）	-17.00%
6	风险分析（自动）	-50.00%	22	安全计划	-15.00%
7	联合应用设计（JAD）	-45.00%	23	Rational 统一过程（RUP）	-15.00%
8	测试驱动开发（TDD）	-45.00%	24	六西格玛（通用）	-12.50%
9	缺陷来源度量	-44.00%	25	净室（Clean-room）开发	-12.50%
10	根本原因分析	-43.00%	26	软件质量保证（SQA）	-12.50%
11	质量功能展开	-40.00%	27	CMM 2	-12.00%
12	CMM 5	-37.00%	28	全面质量保证（TQM）	-10.00%
13	用户参与的敏捷方法	-35.00%	29	未使用 CMM	0.00%
14	风险分析（手工）	-32.00%	30	CMM 1	5.00%
15	CMM 4	-27.00%		平均	-30.12%
16	防错法（Poka-yoke） [⊖]	-23.00%			

第二种方法，即“参与正式审查”具有超过 40 年的经验数据。就缺陷去除效率而言，需求、设计和其他可交付物的正式审查非常有效而且相当高效。除此之外，正式审查的参与者还可以了解到软件缺陷的出现模式和类别，并在他们自己的工作中自发地避免这些缺陷。

一种新兴的风险分析方式非常新颖，目前还缺乏任何相关的经验数据。这种新方法要求在开始软件应用项目之前或在开发上花费任何资金之前进行非常早的规模估算和风险分析。

如果项目风险明显高于项目价值，根本就不启动该项目显然可以预防 100% 的潜在缺陷。澳大利亚维多利亚州政府已经启动了一项这样的计划，通过风险分析并在项目开始之前叫停危险的软件应用项目，他们已经节省了数百万美元资金。

基于模式匹配的新应用规模估算方法可以比以前提前 6 个月进行风险分析。该新方法目前前途光明但需要更多的研究。

还有很多其他事情也会对缺陷预防产生影响。其中之一是为软件人员提供测试或软件质量保证知识的认证。认证还会对缺陷去除具有一定影响。对缺陷预防的影响以负的百分比值表示，而对缺陷去除的影响则以正的百分比值表示。

这里，所示数据也只是粗略估计值，且具体百分数值来自于很少的数据，故不应该作

⊖ Poka-yoke 意为“防误、防错”，亦即 Error & Mistake Proofing。日本质量管理专家、著名的丰田生产体系创建人新江滋生（Shigeo Shingo）根据其长期从事现场质量改进的丰富经验，首创了 POKA-YOKE 的概念，并将其发展成为用以获得零缺陷，最终免除质量检验的工具。——译者注。

为任何事情的判断依据。表 9-18 以缺陷预防进行排序。

表 9-18 认证对缺陷预防和缺陷去除方面的影响

	认证	缺陷预防获益	缺陷去除获益
31	六西格玛绿带	-12.50%	10.00%
32	国际软件测试质量委员会 (ISTQB)	-12.00%	10.00%
33	认证软件质量工程师 (CSQE) - ASQ	-10.00%	10.00%
34	认证软件质量分析师 (CSQA)	-10.00%	10.00%
35	认证软件测试经理 (CSTM)	-7.00%	7.00%
36	六西格玛绿带	-6.00%	5.00%
37	微软认证 (测试)	-6.00%	6.00%
38	认证软件测试专业人士 (CSTP)	-5.00%	12.00%
39	认证软件测试工程师 (CSTE)	-5.00%	12.00%
40	认证软件项目经理 (CSPM)	-3.00%	3.00%
	平均	-7.65%	8.50%

表 9-18 中的数据不是精确值,只是近似值。人们需要对各种认证的有效性进行大量研究。此外,2009 年前后,软件行业有大量的重复和冗余认证。很多软件测试和质量协会都会提供认证,但这些分散组织采用不同方法进行认证,且彼此并不一致。在缺乏统一的协会或认证机构的情况下,各种非营利性或营利性的软件测试和质量保证协会使用完全迥异的标准提供了彼此竞争的认证。

正如本书早先讨论过的,另一组对缺陷预防产生影响的因素是各种度量指标和度量方法。

各种度量指标和度量方法要想对缺陷预防产生影响,它们必须能够表明软件质量等级以及度量相对于软件质量基线的软件质量变化。因此,各种“特性”度量方法和指标被排除在外,因为它们不可测量。

表 9-19 展示了各种度量方法和度量指标对缺陷预防和缺陷去除的大致影响。IFPUG 功能点独占鳌头,因为它可以在需求、设计及代码中对缺陷进行量化。IFPUG 功能点也可用来度量软件缺陷去除成本和软件质量的经济价值。

表 9-19 软件度量指标、度量措施和缺陷预防与缺陷去除

	指标	缺陷预防获益	缺陷去除获益
41	IFPUG 功能点	-30.00%	15.00%
42	六西格玛	-25.00%	20.00%
43	质量成本 (COQ)	-22.00%	15.00%
44	根本原因分析	-20.00%	12.00%
45	TSP/PSP	-20.00%	18.00%
46	月需求变更率	-17.00%	5.00%
47	目标问题 (Goal-question) 指标	-15.00%	10.00%
48	缺陷去除效率	-12.00%	35.00%
49	用例点数	-12.00%	5.00%
50	COSMIC 功能点	-10.00%	10.00%

(续)

	指标	缺陷预防获益	缺陷去除获益
51	控制流程图循环复杂度	-10.00%	7.00%
52	测试覆盖率	-10.00%	22.00%
53	需求遗漏百分比	-7.00%	3.00%
54	故事点数	5.00%	-5.00%
55	平均缺陷成本	10.00%	-15.00%
56	代码行 (LOC)	15.00%	-12.00%
	平均	-11.25%	9.06%

需要注意的是最后两个指标：平均缺陷成本和代码行。如表中所示，它们是有害的度量指标而不是有益指标，它们违反了标准经济学的基本假设。

注意，表 9-19 中排名垫底的两个度量指标对缺陷去除具有负面影响；也就是说，它们会使软件质量变得更加糟糕而不是越来越好。正如软件文献中经常使用的，平均缺陷成本和代码行均非常接近“失职”，它们违反了标准经济学的基本假设，扭曲了最终度量结果。

代码行指标对高级编程语言非常不利，它使得低级编程语言的软件质量和生产率看起来均要比实际的好。此外，这个指标甚至不能用于度量需求和设计缺陷以及任何其他形式的非代码缺陷。

平均缺陷成本对质量也非常不利，它使漏洞百出的应用软件看起来比那些只有少数缺陷的软件在质量上要好很多。这个指标还无法应用到零缺陷的应用软件上。对软件质量非常不利的名义质量度量指标——平均缺陷成本，甚至都无法用于表明最高质量水平，这是软件行业专业失当行为很好的典型例子。

本章讨论的缺陷预防最后一个方面是各种国际标准的有效性。很不幸，国际标准有效性的可用经验数据少之又少。

没有任何已知的对照研究能够证明如果严格遵守国际标准是否会改进软件质量。据有些传闻讲，至少某些国际标准，如 ISO 9001 ~ 9004，会降低软件质量。因为某些没有使用这些国际标准的公司比那些已经通过该标准认证的公司具有更高的质量水平。表 9-20 显示了类似结论。但该表有两个小问题，它只列出了少数国际标准，且其排名基于极少且不完整的信息。

表 9-20 国际标准、缺陷预防和缺陷去除

	标准或政府法规	缺陷预防获益	缺陷去除获益
57	ISO/IEC10181 安全框架	-25.00%	25.00%
58	ISO17799 安全	-15.00%	15.00%
59	《萨班斯-奥克斯利法案》	-12.00%	6.00%
60	ISO/IEC25030 软件产品质量需求	-10.00%	5.00%
61	ISO/IEC9126-1 软件工程产品质量	-10.00%	5.00%
62	IEEE730-1998 软件质量保证计划	-8.00%	5.00%
63	IEEE1061-1992 软件度量指标	-7.00%	2.00%

(续)

	标准或政府法规	缺陷预防获益	缺陷去除获益
64	ISO9000-9003 质量管理	-6.00%	5.00%
65	ISO9001:2000 质量管理体系	-4.00%	7.00%
	平均	-10.78%	8.33%

在软件之外的很多领域,如医疗行业,行业标准通常经过各种临床试验、对照研究及广泛深入的分析论证。而对于软件行业,各种标准并未经过广泛验证且都是基于各种标准委员会自己的主观意见。虽然某些委员会是由知名专家组成,因而其标准可能是有用的,但发布该标准之前缺乏相关验证及现场试验仍是一个信号,其表明软件工程在被视为一个完整的工程学科之前,需要更多的发展和改进。

从表 9-17 到表 9-20,共列举了 65 个缺陷预防方法和实践措施。人们在实践中并不会同时使用它们。表 9-21 显示的是在几百家美国公司(以及 50 家海外公司)中观察到的各种缺陷预防方法和实践措施的大致使用情况。

表 9-21 有点儿小麻烦,因为排名前三的方法已被证明对软件质量是有害的,它们使软件质量更加糟糕,而不是更好。实际上,在那些真正有益的缺陷预防方法中,只有极少数如原型法、测试覆盖率度量 and 联合应用设计(JAD)等在美国有超过 50% 的使用率。

许多最强大、最有效的缺陷预防方法,诸如正式审查或者质量成本度量,其使用率不超过三分之一。表 9-21 中所示数据并不精确,需要更大量的样本数据。然而,这也确实说明有效的软件缺陷预防方法和美国日常实际使用方法之间严重脱节。

表 9-21 软件缺陷预防方法的使用模式

缺陷预防方法	美国项目 使用比率	缺陷预防方法	美国项目 使用比率
1 重用(未认证来源)	90.00%	17 六西格玛	24.00%
2 平均缺陷成本	75.00%	18 风险分析(手工)	22.00%
3 代码行(LOC)	72.00%	19 Rational 统一过程(RUP)	22.00%
4 功能原型	70.00%	20 控制流图循环复杂度	21.00%
5 测试覆盖率	67.00%	21 CMM 1	20.00%
6 未使用 CMM	50.00%	22 月度需求变更率	20.00%
7 联合应用设计(JAD)	45.00%	23 代码复杂度分析	19.00%
8 需求遗漏百分比	38.00%	24 ISO 9001:2000 质量管理体系	19.00%
9 软件质量保证(SQA)	36.00%	25 微软认证(测试)	18.00%
10 用例	33.00%	26 ISO 9000-9003 质量管理	18.00%
11 IFPUG 功能点	33.00%	27 根本原因分析(系统和嵌入式软件)	17.00%
12 测试驱动开发(TDD)	30.00%	28 ISO/IEC 9126-1 软件工程产品质量	17.00%
13 质量成本(COQ)	29.00%	29 TSP/PSP	16.00%
14 Scrum 会议(每日)	28.00%	30 ISO/IEC 25030 软件产品质量需求	16.00%
15 CMM 3	28.00%	31 IEEE 1061-1992 软件度量指标	16.00%
16 用户参与的敏捷	27.00%	32 缺陷来源度量	15.00%

(续)

缺陷预防方法	美国项目 使用比率	缺陷预防方法	美国项目 使用比率
33 根本原因分析(商业和IT软件包)	15.00%	50 认证软件测试专业人士(CSTP)	8.00%
34 IEEE 730-1998 软件质量保证计划	15.00%	51 安全计划	7.00%
35 PSP/TSP	14.00%	52 质量功能展开(QFD)	6.00%
36 软件六西格玛	13.00%	53 全面质量管理(TQM)	6.00%
37 六西格玛(通用)	13.00%	54 认证软件项目经理(CSPM)	6.00%
38 故事点	13.00%	55 国际软件测试质量委员会(ISTQB)	4.00%
39 参与审查	12.00%	56 认证软件质量分析师(CSQA)	4.00%
40 CMM 2	12.00%	57 认证软件测试工程师(CSTE)	4.00%
41 《萨班斯-奥克斯利法案》	12.00%	58 COSMIC 功能点	4.00%
42 六西格玛绿带	11.00%	59 认证软件质量工程师(CSQE)-ASQ	3.00%
43 ISO/IEC 10181 安全框架	11.00%	60 风险分析(自动)	2.00%
44 六西格玛黑带	10.00%	61 认证软件测试经理(CSTM)	2.00%
45 缺陷去除效率	10.00%	62 重用(认证来源)	1.00%
46 用例点数	10.00%	63 CMM 5	1.00%
47 ISO 17799 安全	10.00%	64 防错法(Poka-yoke)	0.10%
48 目标问题指标	9.00%	65 净室软件开发	0.10%
49 CMM 4	8.00%		

这种令人沮丧的现实状况，部分原因是因为缺陷预防方法的实际度量和研究相当困难。只有少数大型成熟的软件企业才有能力进行缺陷预防研究。甚至大多数大学都无法开展缺陷预防研究，因为它们缺乏同大公司足够多的接触，因此没有什么可用数据。

总而言之，软件缺陷预防在软件文献中涉及极少，也很少有可靠的经验数据可用，因而在该主题上，软件行业仍需大量研究。

改进缺陷预防方法和缺陷去除活动的一种方法是，创建一个非营利的基金会或者协会，广泛地研究各种软件质量主题。这将既包括缺陷预防，也包括缺陷去除。以下是笔者提议成立的非营利性国际软件质量基金会(ISQF)的假设机构和职能。

9.4.1 非营利性国际软件质量基金会(ISQF)的提议

国际软件质量基金会(International Software Quality Foundation, ISQF)将是一个非营利性的基金会，致力于提高软件应用产品的质量和经济价值。该基金会的法人组织形式由最初的董事会决定，其目的是依据“国内税收法典”第501(C)章来成立一个公司，从而成为一个有资格接受捐赠的免税组织。

国际软件质量基金会(ISQF)将遵循的基本原则如下：

1. 低劣软件质量已经并且正在破坏软件社区的专业声誉。
2. 低劣软件质量已经并且正在导致客户和软件开发组织之间的大量诉讼案件。
3. 显著的软件质量改善在技术上是可行的。

4. 软件质量改善在降低软件成本和保证开发进度方面具有可观的经济效益。

5. 软件质量改善取决于多种形式的准确质量度量,包括但不限于:度量软件缺陷数目、软件缺陷来源、软件缺陷严重性级别、缺陷预防方法、缺陷去除方法、客户满意度以及软件团队士气。

6. 软件开发和维护的主要成本用于消除缺陷。ISQF 将会把主要研究方向放在度量缺陷预防、缺陷去除和客户满意度的经济价值上。

7. 软件度量和软件评估是相辅相成的技术。ISQF 将对软件质量和可靠性评价方法进行评估,并公布评估结果。该评估将根据标准基准研究和测试用例独立进行,不接受软件估算工具厂商的任何费用资助。

8. 软件缺陷可能来自需求、设计、编码、用户文档以及测试计划和测试用例本身。此外,当试图修复较早发现的缺陷时,也可能会引入次生缺陷。ISQF 将研究软件问题出现的所有来源并试图对软件缺陷和导致用户不满的所有根源进行改进。

9. ISQF 将发起在某些技术主题上的研究,这些技术主题包括但不限于:非正式审查、静态分析、测试用例设计、测试覆盖率分析、测试工具、缺陷报告、缺陷跟踪工具、不良修复注入、易错模块去除、复杂度分析、缺陷预防、正式审查、质量度量方法及质量度量指标。

10. ISQF 也会赞助对影响软件质量的所有社会因素进行研究,以量化这些因素的影响。这包括软件质量保证(SQA)组织的有效性,独立测试机构、独立软件维护机构、国际标准以及软件认证的价值。同样,还会支持研究软件客户满意度的方法。

11. 信息技术基础架构库(ITIL)中定义的服务标准取决于所达到的软件质量满意水平。就交付缺陷而言,ISQF 将会吸纳 ITIL 库中的原则,并赞助可靠性及可用性与质量水平之间相互关系的研究。

12. 随着软件行业的新技术不断出现,与这些新技术对软件质量的影响保持同步非常重要。ISQF 将会独自进行或委托相关机构或组织开展各种新方法对质量结果影响的研究,这些新方法包括但不限于:敏捷开发、云计算、水晶开发方法、极限编程、开源应用开发以及面向服务架构等。

13. ISQF 将对大学教育提供示范课程,包括软件度量方法、度量指标、缺陷预防、缺陷去除、客户支持、客户满意度和软件质量的经济价值。

14. ISQF 将向 MBA 项目提供涉及软件经济学和软件管理原则的示范课程。质量经济学将是主要的子主题。

15. ISQF 将向企业提供示范课程及内部培训,包括:软件度量方法、度量指标、缺陷预防、缺陷去除、客户支持、客户满意度及软件质量的经济价值。

16. ISQF 也会提供软件质量保证(SQA)、软件测试、软件客户支持和软件质量度量等职业的推荐技能概要介绍。

17. ISQF 将提供软件质量保证(SQA)、软件测试、软件客户支持和软件质量度量等职业的认证考试并授予认证证书。其中,软件质量度量当前还没有任何认证。

18. ISQF 将建立一个能力胜任的理事会来管理认证考试并定义最先进的软件质量保证

(SQA)、软件测试和软件质量度量方法。未来可能会增加其他理事会或专业。

19. ISQF 会定义适用于软件质量控制方法不足时的失职 (malpractice) 条款。其示例包括：未能保持足够的软件缺陷记录、未能使用足够的测试阶段和测试用例以及未能对关键材料执行足够的审查。

20. ISQF 将会和其他关注类似问题的非营利性组织进行合作。这些组织包括但不限于：位于比利时的全球软件质量协会 (GASQ)、世界质量大会、IEEE、ISO、ANSI、IFPUG、软件过程改进网络 (SPIN)^①以及 SEI。ISQF 也和其他组织合作，比如各个大学、信息技术指标与生产力研究所 (ITMPI)、项目管理协会 (PMI)、质量保证协会 (QAI)、软件测试协会，以及相关的工程、基准研究和专业组织 (比如 ISBSG 基准研究组织)。ISQF 还会与海外类似的质量组织进行合作，比如中国、日本、印度及俄罗斯的那些质量组织。这种合作可能包括互认会员资格，如果这些组织愿意以这种方式参与合作。

21. ISQF 将由一个由会员选举出来的 5 名主管组成的理事会负责管理。主管理事会将任命一位主席和一个首席执行官 (CEO)。主席将任命一名会计、秘书及其他注册地法律要求的工作人员。起初，理事会、主席及工作人员以志愿者身份提供无偿服务。为确保吸纳新鲜力量，主席任期 2 年。

22. ISQF 的资金由会费、捐赠、资助以及筹款活动，如会议和活动等共同筹集。

23. ISQF 还会有一个由 5 名成员组成的技术顾问理事会，其成员由主管理事会主席任命。这个顾问理事会将协助 ISQF 保持最前沿的研究主题，这些主题包括：软件测试、正式审查、质量度量指标、可用性和可靠性以及其他 ITIL 指标。

24. ISQF 将使用现代通信方法以扩展质量相关主题的信息传播。这些通信方法包括一个 ISQF 网站、网络研讨会、质量主题的维基百科 (Wikipedia)、Twitter、博客以及在线时事通讯等。

25. ISQF 将会拥有一些分委员会，来处理诸如会员、资助和捐赠、新闻联络、大学联络及与其他非营利性组织 (如比利时的全球软件质量协会) 的联络。

26. 为提高人们对质量重要性的认识，ISQF 将会出版一个季刊杂志，暂定名称为《软件质量进展》(Software Quality Progress)。这是一个匿名审稿的杂志，全部审稿人均来自 ISQF 会员。

27. 为提高人们对软件质量重要性的认识，ISQF 将会发起一个年度大会，并征集一系列的“杰出质量奖”提名。初始奖项将以软件类型来设置，如信息系统、商业应用软件、军用软件、外包应用软件、系统与嵌入式软件、Web 应用软件等。这些奖项将授予那些具有最低交付缺陷数目、最高缺陷去除效率水平、最好客户服务和最高客户满意度排名的团队、公司或组织。

28. 为提高人们对软件质量重要性的认识，ISQF 将鼓励其会员撰写和评论软件质量主

① 软件过程改进网络，Software Process Improvement Network，简称 SPIN。软件过程改进网络是由有志于软件和系统过程改进的专业人士组成的松散组织。目前全球各地有 114 个 SPIN 组织，每个 SPIN 组织完全独立。SEI 对 SPIN 提供支持，包括创建、维护和分发 SPIN 目录，联络新出现或现有 SPIN 的软件专业人士，以及发布 SPIN 活动信息。详情请参见 <http://www.sei.cmu.edu/spin/>。——译者注

题的文章和书籍。技术杂志（比如《CrossTalk》）和主流商业杂志（比如《哈佛商业评论》）均为可选杂志。

29. 为提高人们对软件质量重要性的认识，ISQF 将开始收集并建立一个以软件质量相关主题和问题的书籍、杂志文章和论文专著为主的图书馆。

30. 为提高人们对软件质量重要性的认识，ISQF 将发起软件缺陷、缺陷严重性等级、缺陷去除效率、测试覆盖率、审查效率、审查和测试成本、质量成本（COQ）以及那些因糟糕质量而遭原告投诉的软件诉讼案件等的基准研究。

31. 为提高人们对糟糕软件质量所引起经济后果的认识，ISQF 将会发起与较差软件质量相关联的间接损坏、死亡及财产损失的研究。

32. 为提高人们对糟糕软件质量所引起经济后果的认识，ISQF 将收集那些因软件质量低劣而导致的诉讼案件结果的公开信息。这些诉讼中，软件质量不佳是原告提起索赔的部分原因。这些诉讼包括违反合同、诈骗以及低劣软件质量损害客户业务经营等案件。

33. 为提高人们对软件质量重要性的认识，ISQF 将鼓励其分会保持在州级或地区级别，比如罗德岛软件质量协会或波士顿软件质量协会。

34. 为确保高标准的软件质量教育，ISQF 将会审查和认证由大学及私人公司提供的具体软件质量课程材料。所有提交课程材料的认证均基于自愿原则。每个认证将收取最低限度费用以支付相关支出。无论课程通过与否都将根据时间和材料花费而征收这些费用。

35. 为确保在合同或外包协议中包含并正确定义软件质量议题，ISQF 将在软件质量的法律地位与合同问题上，与美国律师协会、州律师协会、美国仲裁协会以及各种法律学校展开合作。

36. ISQF 会员将被要求完全同意并遵守由 ISQF 技术顾问理事会规定的全部道德准则。该道德准则将包括诸如向所有咨询者提供完整真实的质量信息、避免利益冲突、基于确凿可靠经验信息给出质量建议等要求。

37. 因为软件安全和软件质量紧密相关，ISQF 还将安全攻击预防及从安全攻击中恢复等方面主题作为自己全部使命的一部分。但是，软件安全高度专业化，它要求质量保证人员和软件测试人员正常培训之外的额外技能。

38. 由于严重的全球经济衰退，ISQF 将试图快速传播软件质量经济价值的经验数据。高质量软件已经被证明可以缩短开发周期、降低开发成本、提高客户支持度、降低维护成本。但仅有少数软件项目经理和高管能够接触到支持上述主张的详细数据。

软件工程和软件质量需要前所未有地紧密结合。更加广泛地推广应用更好的质量预测方法、更优的质量度量措施、更加有效的缺陷预防方法和缺陷去除方法，都与推动软件工程向真正的工程学科之地位前进的大方向完全一致。

9.5 软件缺陷去除

软件缺陷预防和软件缺陷去除都很重要，但研究和度量缺陷去除则较为容易些。这是因为，正式审查、静态分析和软件测试中发现的缺陷数量为计算缺陷去除效率水平提供了

一种定量基础。

虽然在理论上研究软件缺陷去除更容易是事实，但当前相关的文献资料仍然稀缺得令人沮丧。例如，软件测试专业拥有大量的文献资料、数以百计的书籍、数以千计的期刊文章、许多专业协会以及为数众多的技术会议。但是，几乎还没有任何测试文献包含创建的测试用例数目、实际发现和去除的缺陷数目、不良修复注入率数据或者其他确凿的数据点等经验数据。

在所有软件测试文献中几乎根本没有涉及某些重要主题。例如，IBM 做过的一项研究曾发现，测试用例中的错误比这些测试用例测试的软件中的错误还多。同样的研究还发现了 35% 的重复或冗余测试用例。然而，几乎所有的软件测试文献中都既没有讨论测试用例错误，也没有讨论这类冗余测试用例。

软件测试和其他缺陷去除文献的另一个不足与不良修复注入有关。大约 7% 的软件缺陷修复自身含有新缺陷。实际上，有时会有次生的、甚至第三重的不良修复，也就是说，连续三次试图修复一个缺陷，可能还是没能修复最初的软件缺陷，却意外引入了以前没有的新缺陷。

软件工程文献和软件专业协会的另一个问题是专业关注点太窄。大多数软件测试组织经常忽视静态分析或正式审查。

关注点狭窄的后果是，各种缺陷去除方法的协同作用在软件质量与软件工程文献中没有很好地涉及。例如，执行需求和设计的正式审查不仅可以发现缺陷，而且可以为构建测试用例提供更好且更加完整的源材料，从而提升后续测试阶段至少 5% 的缺陷去除效率。

测试之前运行自动化静态分析将会发现众多与限定条件、边界条件以及结构化问题相关的缺陷，从而加速后续测试工作。

在发现那些需要人类智慧和洞察力、非常复杂且微妙的软件问题上，正式审查是最好的方法。正式审查在寻找软件需求及设计的遗漏和歧义性方面也是最好的方法。

静态分析在寻找结构性和机械性问题方面表现最佳，比如边界条件、代码重复、错误处理失效、不正确的例程分支等。静态分析还可以发现安全漏洞。

软件测试在发现那些只有软件运行时才会出现的问题上表现最好，比如性能问题、可用性问题和安全性问题等。

单独来看，这三种方法都非常有用但都不完整。当一起使用时，其协同效应可以提高缺陷去除效率水平，还可以降低与缺陷去除活动相关的工作量和项目成本。

表 9-22 列出了 80 种不同形式的软件缺陷去除方法：静态分析、审查、各种测试方法以及许多与软件诉讼相关的特殊形式缺陷去除方法。

尽管表 9-22 显示了缺陷去除效率的整体价值，但这些数据也确实涉及了选定缺陷分类的缺陷去除效率。例如，自动化静态分析可以发现 87% 的结构化代码问题，但它无法发现需求遗漏或者诸如千年虫等这种来自需求的软件问题。

表 9-22 80 种软件缺陷去除活动概况

缺陷去除活动				
	活动	每功能点测试用例数	缺陷去除效率	不良修复注入百分比
静态分析				
1	自动化静态分析	0.00	87.00%	2.00%
2	需求审查	0.00	85.00%	6.00%
3	外部设计审查	0.00	85.00%	6.00%
4	用例审查	0.00	85.00%	4.00%
5	内部设计审查	0.00	85.00%	4.00%
6	新代码审查	0.00	85.00%	4.00%
7	重用认证审查	0.00	84.00%	2.00%
8	测试用例审查	0.00	83.00%	5.00%
9	自动化文档分析	0.00	83.00%	6.00%
10	遗留代码审查	0.00	83.00%	6.00%
11	质量功能展开	0.00	82.00%	3.00%
12	文档校对 (Proof reading)	0.00	82.00%	1.00%
13	本地化审查	0.00	81.00%	3.00%
14	软件架构审查	0.00	80.00%	3.00%
15	测试计划审查	0.00	80.00%	5.00%
16	测试脚本审查	0.00	78.00%	4.00%
17	测试覆盖率分析	0.00	77.00%	3.00%
18	文档编辑	0.00	77.00%	2.50%
19	结对编程评审	0.00	75.00%	5.00%
20	六西格玛分析	0.00	75.00%	3.00%
21	缺陷修复审查	0.00	70.00%	3.00%
22	业务规划审查	0.00	70.00%	8.00%
23	根本原因分析	0.00	65.00%	4.00%
24	治理评审	0.00	63.00%	5.00%
25	代码重构	0.00	62.00%	5.00%
26	易错模块分析	0.00	60.00%	10.00%
27	独立审计	0.00	55.00%	10.00%
28	内部审计	0.00	52.00%	10.00%
29	Scrum 会议 (每日)	0.00	50.00%	2.00%
30	质量保证评审	0.00	45.00%	7.00%
31	《萨班斯-奥克斯利法案》评审	0.00	45.00%	10.00%
32	用户故事评审	0.00	40.00%	10.00%
33	非正式结对评审	0.00	40.00%	10.00%
34	独立验证和确认	0.00	35.00%	12.00%
35	个人桌面检查	0.00	35.00%	7.00%
36	阶段评审	0.00	30.00%	15.00%

(续)

缺陷去除活动				
	活动	每功能点测试用例数	缺陷去除效率	不良修复注入百分比
37	正确性证明	0.00	27.00%	20.00%
	平均	0.00	66.92%	6.09%
常规测试				
38	PSP/TSP 单元测试	3.50	52.00%	2.00%
39	子程序测试	0.25	50.00%	2.00%
40	极限编程测试	2.00	40.00%	3.00%
41	组件测试	1.75	40.00%	3.00%
42	系统测试	1.50	40.00%	7.00%
43	新功能测试	2.50	35.00%	5.00%
44	回归测试	2.00	30.00%	7.00%
45	单元测试	3.00	25.00%	4.00%
	平均	2.06	41.00%	4.13%
	总和	16.50		
自动化测试				
46	病毒 / 间谍软件测试	3.50	80.00%	4.00%
47	系统测试	2.00	40.00%	8.00%
48	回归测试	2.00	37.00%	7.00%
49	单元测试	0.05	35.00%	4.00%
50	新功能测试	3.00	35.00%	5.00%
	平均	2.11	45.40%	5.60%
	总和	10.55		
特种测试				
51	病毒测试	0.70	98.00%	2.00%
52	间谍软件测试	1.00	98.00%	2.00%
53	安全测试	0.40	90.00%	4.00%
54	极限 / 容量测试	0.50	90.00%	5.00%
55	安全渗透测试	4.00	90.00%	4.00%
56	可重用性测试	4.00	88.00%	0.25%
57	防火墙测试	2.00	87.00%	3.00%
58	性能测试	0.50	80.00%	7.00%
59	本地化测试	0.30	75.00%	10.00%
60	可扩展性测试	0.40	65.00%	6.00%
61	平台测试	0.20	55.00%	5.00%
62	净室测试	3.00	45.00%	7.00%
63	供应链测试	0.30	35.00%	10.00%
64	SOA 业务流程测试	0.20	30.00%	5.00%
65	独立测试	0.20	25.00%	12.00%

(续)

缺陷去除活动				
	活动	每功能点测试用例数	缺陷去除效率	不良修复注入百分比
	平均	1.18	70.07%	5.48%
	合计	17.70		
用户测试				
66	可用性测试	0.25	65.00%	4.00%
67	本地化测试	0.40	60.00%	3.00%
68	实验室测试	1.25	45.00%	5.00%
69	外部 Beta 测试	1.00	40.00%	7.00%
70	内部验收测试	0.30	30.00%	8.00%
71	外包验收测试	0.05	30.00%	6.00%
72	COTS 验收测试	0.10	25.00%	8.00%
	平均	0.48	42.14%	5.86%
	总和	3.35		
诉讼分析、测试				
73	知识产权测试	2.00	80.00%	1.00%
74	知识产权评审	0.00	80.00%	3.00%
75	违反合同评审	0.00	80.00%	2.00%
76	违反合同测试	2.00	70.00%	2.00%
77	税务诉讼评审	0.00	80.00%	4.00%
78	税务诉讼测试	1.00	70.00%	4.00%
79	欺诈代码评审	0.00	80.00%	2.00%
80	代码剽窃评审	0.00	80.00%	2.00%
	平均	2.35	77.14%	2.71%
	总和	5.00		
	每功能点测试用例总数	53.10		

表 9-22 以缺陷去除效率降序排列, 所示结果均为最大值。在实际软件项目中, 缺陷去除效率的度量范围也许小于表 9-22 中所示最大标称值的一半。

虽然表 9-22 中列出了 80 种不同的软件缺陷去除活动, 但这并不意味着在实际软件项目中会同时实施这些活动。实际上, 美国普遍的缺陷去除活动仅仅包括以下 6 种测试:

美国软件缺陷去除的平均序列

1. 单元测试
2. 新功能测试
3. 性能测试
4. 回归测试
5. 系统测试
6. 验收或 Beta 测试

整体上，这 6 种测试形式的累积缺陷去除效率水平范围在 70% 到 85% 之间，远低于高可靠性和客户满意度所要求的水平。基本上，单靠软件测试这一种缺陷去除活动并不能实现卓越的软件质量水平。

软件缺陷去除活动的最佳序列将包括几种测试之前的正式审查、静态分析和至少 8 种测试形式：

软件缺陷去除的最优序列

测试之前的缺陷去除活动

1. 需求审查
2. 架构审查
3. 设计审查
4. 代码审查
5. 测试用例审查
6. 自动化静态分析

测试缺陷去除

7. 子程序测试
8. 单元测试
9. 新功能测试
10. 安全测试
11. 性能测试
12. 可用性测试
13. 系统测试
14. 验收或 Beta 测试

在每一个软件项目中，软件缺陷去除活动的上述协同组合将使累计缺陷去除效率水平超过 95%，有些项目可以达到 99%。

当最有效的缺陷去除方法与最有效的缺陷预防方法相结合时，软件工程将能够实现始终如一的卓越质量水平。软件项目广泛应用这种组合并使之成为常态之时，就是软件工程成为真正的工程学科之日。

9.6 软件质量专家

正如本书前面曾提到的，软件工程领域具有超过 115 种职业和专家。在大多数知识密集型行业，比如医药和法律行业，专家们经过特殊的培训，这些技能使得他们在所从事的领域，如神经外科或海事法，远远胜过那些通才。

对于软件工程，关于专家角色方面的文献极其缺乏，即使那些仅有的文献资料也常常是含混不清、模棱两可的。在关于软件专业化的文献中，很多都写得云山雾罩的，但却仅仅是表达了那些作者自己的某些偏好。而很多文献作者更喜欢一种通才模型。在这种模型里，每个团队成员个体都是可以互换的，如果需要，每个个体都可以从事需求、设计、开

发和软件测试等不同工作。而另外一些作者则更倾向于专家模型。在这种模型中，诸如测试、质量保证和软件维护等具有关键技能的工作，是由经过专门培训的专家来完成的。

本章，我们将主要关注两个基本问题：

1. 专门技能能够降低潜在缺陷及有益于缺陷预防吗？
2. 专门技能能够提升缺陷去除效率水平吗？

并非所有 115 种专家都会在这里讨论。就软件潜在缺陷和缺陷预防而言，我们将会讨论那些对软件质量水平具有潜在影响的角色。

本章将讨论如下所列的 20 种专家（以字母顺序排序）：

1. 软件架构师
2. 业务分析师
3. 数据库分析师
4. 数据质量分析师
5. 企业架构师
6. 软件估算专家
7. 功能点专家
8. 审查会议主持人
9. 软件维护专家
10. 需求分析师
11. 软件性能专家
12. 风险分析专家
13. 软件安全专家
14. 六西格玛专家
15. 系统分析师
16. 软件质量保证 (SQA)
17. 技术文档作家
18. 软件测试员
19. 可用性专家
20. Web 设计师

对这 20 种专家的每一个，我们会考虑他们所面临的潜在缺陷数量，以及他们是否会对软件缺陷预防和缺陷去除活动产生实际影响。

表 9-23 根据每个专家所需要负责的任务范围对这些专家进行了排名。“任务范围”这个指标代表了正常分配给一个从业者的功能点数量。表 9-23 也显示了作为其工作一部分，各种职业所面对的缺陷数量。表 9-23 还显示了这些专业职业在缺陷预防和缺陷去除上的大概影响。

排名靠前的专家面临着大量的潜在缺陷，这些缺陷可能会使整个公司蒙受极大损失，对其所拥有的软件也是一样。下面将简要讨论这 20 种专家。

表 9-23 软件专家对软件质量的影响

	专业化职业	任务范围	潜在缺陷	缺陷预防	缺陷去除
1	风险分析专家	300 000	7.00	75.00%	25.00%
2	企业架构师	250 000	6.00	25.00%	20.00%
3	六西格玛专家	250 000	5.00	25.00%	30.00%
4	数据库分析师	100 000	3.00	15.00%	10.00%
5	软件架构师	100 000	3.00	17.00%	12.00%
6	可用性专家	100 000	1.00	10.00%	15.00%
7	软件安全专家	50 000	7.00	70.00%	20.00%
8	数据质量分析师	50 000	5.00	12.00%	15.00%
9	业务分析师	50 000	3.50	25.00%	10.00%
10	软件估算专家	25 000	3.00	20.00%	25.00%
11	系统分析师	20 000	6.00	20.00%	20.00%
12	软件性能专家	20 000	1.00	10.00%	12.00%
13	软件质量保证专家	10 000	5.50	15.00%	40.00%
14	Web 设计师	10 000	4.00	15.00%	12.00%
15	需求分析师	10 000	4.00	20.00%	15.00%
16	软件测试员	10 000	3.00	15.00%	50.00%
17	功能点专家	5000	4.00	10.00%	10.00%
18	技术文档作家	2000	1.00	10.00%	10.00%
19	软件维护专家	1500	3.50	30.00%	20.00%
20	审查会议主持人	1000	4.50	27.00%	35.00%
	平均	68 255	4.00	23.30%	20.30%

9.6.1 风险分析专家

□ 任务范围 = 300 000 功能点

□ 潜在缺陷 = 7.00

□ 缺陷预防影响 = -75%

□ 缺陷去除影响 = 25%

30 万个功能点的大任务范围意味着这些公司并不需要很多风险分析师，但是那些已雇用的风险分析师必须非常能干，既要非常了解技术风险，又要非常熟悉财务风险。自 2008 年以来，由于全球经济衰退的发生，导致大量企业倒闭。很明显，这些企业的风险分析还没有做到它应该达到的成熟程度，尤其是在应对财务风险上。

风险分析师面临着与任何特定软件应用程序相关联、超过 100% 的潜在缺陷。他们确实既需要应对技术风险和质量风险，又需要解决财务风险和法律风险，而这些都是软件质量和缺陷度量等正常软件领域之外的领域。

在向主要软件应用项目提供资金之前，正式而谨慎的风险分析可以在任何一大笔钱花出去之前，叫停风险过高的项目投资。对于有问题的项目，在项目启动之前，正式而谨慎的风险分析可以在承诺资金之前要求引进更好的技术。

较早风险分析成功的关键包括：及早调整项目大小的能力、及早成本估算的能力、及早质量评估的能力，以及众多来自以前失败和成功项目分析的潜在风险经验和教训知识。

就软件质量而言，风险分析师的主要职责是，通过风险分析，在不良项目启动之前即加以阻止，以及确保那些使用最顶尖质量方法的项目成功启动。风险分析师也需要理解软件项目失败的主要原因，并且还需要熟悉了解某些软件诉讼案件的最终结果。这些案件包括取消项目、违反合同、盗窃知识产权、专利侵权、通过软件挪用公款、欺诈、税务问题、《萨班斯-奥克斯利法案》问题以及其他形式的诉讼。

9.6.2 企业架构师

- 任务范围 = 250 000 功能点
- 潜在缺陷 = 6.00
- 缺陷预防影响 = -25%
- 缺陷去除影响 = 20%

企业架构师是企业的关键角色之一，其工作职责是理解企业业务的各个方面，整合公司整体项目投资组合以满足公司业务发展的需要，这些项目组合可能包括了 3000 多个独立的软件应用及总数超过上千万个功能点。企业架构师不仅需要了解内部软件的知识领域，还需要了解开源应用和商业软件包（如 Vista 和 SAP）相关的知识领域。

就软件质量而言，企业架构师的主要职责是，理解软件质量对公司经营的业务价值，确保高层管理人员也有类似的理解。企业架构师和公司高层均需不断推动企业卓越发展，以实现既定的软件交付速度。

通过确保未来不会发生诸如千年虫这样的严重错误，企业架构师也在公司治理方面担负着重要职责。

9.6.3 六西格玛专家

- 任务范围 = 250 000 功能点
- 潜在缺陷 = 5.00
- 缺陷预防影响 = -25%
- 缺陷去除影响 = 20%

六西格玛专家的巨大任务范围表明，他们的工作本质上是整个公司层面的，而非仅限于某个特定应用领域。六西格玛专家在质量方面的主要职责是，提供缺陷来源和缺陷成因的专家分析，建议有效的持续改进方法以减少软件错误的主要源头。

9.6.4 数据库分析师

- 任务范围 = 100 000 功能点
- 潜在缺陷 = 7.00
- 缺陷预防影响 = -75%
- 缺陷去除影响 = -25%

现代世界，各大主要公司和政府机构拥有的各种数据比其所拥有的软件还要多。客户数据、员工数据、制造业数据等总量达数以百万计的各种数据记录分散在几十个数据库和资料库中。这种数据集是企业的宝贵财富，在制定关键业务决策时需要访问这些数据，因而需要对这些数据加以保护，防止黑客攻击、盗窃和非授权访问。

当前，没有“数据点”（data point）之类的度量指标来表示数据库和资料库的规模，这可算得上是数据质量领域的重大不足之一。其后果是，数据质量非常难以量化。实际上，数据质量文献根本就没有“数据错误总数”之类的量化数据。

结果，数据库分析师和数据质量分析师面临着重重障碍。在数据质量方面，他们均扮演着重要角色，但缺乏很好完成工作所必要的各种工具。

在软件质量方面，数据库分析师所扮演的重要角色是，确保以最优化方式设计和组织数据库和资料库，以及确保实施和执行各种质量保证过程，以验证企业数据所有数据元素的精确性。

9.6.5 软件架构师

- 任务范围 = 100 000 功能点
- 潜在缺陷 = 3.00
- 缺陷预防影响 = -17%
- 缺陷去除影响 = 12%

软件架构师也拥有巨大的任务范围，他们需要能够预见和应对现代世界各种已知的大型应用软件，比如 Vista、SAP 和 Oracle 等的 ERP 软件包、空中交通管制、国防应用软件以及主要商业应用软件。

在过去 50 年里，应用软件逐步演化，已经从最初在硬件上直接独自运行，逐步演变为在操作系统下运行、作为多层网络一部分而运行以及真正分段分散地运行在云平台上，而该云平台的硬件和软件设施可能分散位于相隔数千里的地方。

当前，软件架构师的职责变得比 10 年前更加复杂。软件架构师需要理解各种现代应用实践，比如面向服务架构（SOA）、云计算、多层结构等。此外，架构师还需要知道各种可重用材料的来源和认证方法，这些材料占到当前许多大型应用软件代码的 50% 以上。

在软件质量方面，软件架构师扮演的主要角色是，理解复杂、多层、高分布式环境中软件缺陷的影响，这种环境中的软件组件可能来自数十个不同来源。

9.6.6 可用性专家

- 任务范围 = 100 000 功能点
- 潜在缺陷 = 1.00
- 缺陷预防影响 = -10%
- 缺陷去除影响 = 15%

术语“可用性”定义了客户需要做些什么才能成功操作软件，它还包括当软件行为不对时客户该怎么办。

可用性专家往往都有认知心理学背景，并且精通各种软件接口：键盘命令、按钮、触摸屏、语音识别，甚至更多。

在软件质量方面，可用性专家的主要职责是，确保应用软件的接口和控制序列尽可能自然和直观。软件开发期间，他们通常与使用该软件的志愿客户一起进行可用性研究。

大型计算机和软件公司（比如 IBM 和微软）拥有可用性实验室。在这里，当客户使用软件和硬件产品的预发布版本时，可以对客户的使用情况进行观察。这些实验室监控键盘按键、屏幕触摸、语音命令和其他接口方法。可用性专家还会在每次客户使用完该软件之后，向客户询问情况，以找出客户喜欢和不喜欢的接口及命令序列。

9.6.7 软件安全专家

- 任务范围 = 50 000 功能点
- 潜在缺陷 = 7.00
- 缺陷预防影响 = -70%
- 缺陷去除影响 = 20%

随着软件安全威胁的不断升级和变化，对更多软件安全专家的需求也不断增长。而在大学和就业后对软件安全专家提供更好培训的需求，也逐渐增多。

截至 2009 年，部分由于经济衰退的原因，软件安全攻击和数据盗窃在数量和复杂性方面快速增长。黑客正在从个体爱好者的单打独斗快速转变为有组织犯罪，甚至能够针对敌对的外国政府发起攻击。

即使安装了主动式反病毒和反间谍软件，处于防火墙之后的应用软件也并不是绝对安全。目前迫切需要依靠使用诸如谷歌的 Caja、E 编程语言以及改变权限模式等技术提高应用软件的安全免疫水平。

软件安全和软件质量并不等同，但它们彼此非常相似，且其预防方法和去除方法也颇为一致且具有协同作用。错误处理程序是应用软件攻击的主要途径这一事实恰恰说明了软件质量和软件安全之间的紧密性。

在软件质量方面，软件安全专家的主要职责是，与最新类型的软件威胁保持同步，确保新应用和遗留应用均具有最顶尖水平的安全防护措施。

9.6.8 数据质量分析师

- 任务范围 = 50 000 功能点
- 潜在缺陷 = 5.00
- 缺陷预防影响 = -12%
- 缺陷去除影响 = 15%

截至 2009 年，数据质量分析师的人数很少，且其使用的工具和技术装备远远不足。目前还没有针对数据数量的有效规模度量指标（比如类似于功能点的“数据点”指标）。因此，在诸如数据库和资料库的潜在缺陷、有效缺陷去除方法，缺陷评估或者缺陷度量等方面上，没有任何可用的经验数据。

理论上，数据质量分析师的职责是预防数据错误发生，推荐有效的数据错误去除方法。然而，鉴于在公共记录、信用评价、会计、税务等方面大量明显的错误，很显然，数据质量甚至落后于软件质量。事实上，在质量控制方面，数据和软件似乎均落后于任何其他工程和技术领域。

9.6.9 业务分析师

- 任务范围 = 50 000 功能点
- 潜在缺陷 = 3.50
- 缺陷预防影响 = -25%
- 缺陷去除影响 = 10%

很多 IT 组织里，业务分析师是软件开发社区和软件用户社区的首要联系中枢。业务分析师必须既非常精通业务需求，又熟悉软件工程技术。

在软件质量方面，业务分析师应该扮演的主要角色是，说服商业和技术社区，使其确信高层次的软件质量可以缩短开发周期、降低开发成本。商业客户常常随意设定时间表，然后试图强迫软件社区依靠缩减正式审查和删减测试的方法去努力满足这些时间表。

好的业务分析师应该具有来自各种来源（比如国际软件基准组织（ISBSG））的可用数据，这些数据显示了软件质量与进度、成本之间的关系。业务分析师还应该理解诸如联合应用设计（JAD）、质量功能展开（QFD）和需求审查等方法的价值。

9.6.10 软件估算专家

- 任务范围 = 25 000 功能点
- 潜在缺陷 = 3.00
- 缺陷预防影响 = -20%
- 缺陷去除影响 = 25%

雇用软件估算专家是一个公司成熟的标志。通常，这些软件估算专家在项目办公室或者某个特殊的员工小组工作，对那些通常没有受过良好估算培训的一线经理们提供软件估算方面的技术支持。

估算专家应该有权获取并熟悉如何使用能够预测软件质量、项目进度和成本的主要软件估算工具，包括 COCOMO、KnowledgePlan、Price-S、SoftCost、SEER、Slim 等工具。实际上，很多公司在同一个软件应用项目中使用多个软件估算工具，以获得准确的估算结果。

在软件质量方面，软件估算专家的主要职责是，尽早预测软件质量。理想情况下，在花费大量资金之前就应该预测软件质量。不仅如此，软件估算专家可能还需要使用多种估算方法以表明各种软件开发实践（比如敏捷开发、团队软件过程（TSP）、Rational 统一过程（RUP）、正式审查、静态分析以及各种测试方法）对软件质量的影响。

9.6.11 系统分析师

- 任务范围 = 20 000 功能点

- 潜在缺陷 =6.00
- 缺陷预防影响 =-25%
- 缺陷去除影响 =20%

软件系统分析员是软件工程或者编程社区与软件最终用户之间的接口点之一。系统分析员和业务分析员承担着相似的职责，但“系统分析员”职位的出现更多是为了那些嵌入式软件或系统软件，这些软件通常是为了技术目的而不是为满足本地商业需求而开发的。

在软件质量方面，系统分析员的主要职责是，理解所有形式的软件表现方法（用户故事、用例、正式规格说明语言、流程图、N-S 图等）都可能含有错误。通过软件测试可能无法发现这些错误，而在任何情况下当测试发现这些错误时都有点儿太晚。因此，系统分析员的一个关键职责是，参加需求、内部设计文档及外部设计文档的正式审查。如果应用软件是由测试驱动开发方法构建的，系统分析员还将参与到测试用例的设计和构建上。系统分析员也将参与诸如联合应用设计（JAD）和质量功能展开（QFD）等活动。

9.6.12 软件性能专家

- 任务范围 =20 000 功能点
- 潜在缺陷 =1.00
- 缺陷预防影响 =-10%
- 缺陷去除影响 =12%

“性能专家”的职位通常只在那些构建非常大型、复杂软件应用、非常大型的公司里才能找到，比如 IBM、Raytheon、Lockheed、波音、SAP、Oracle、Unisys、谷歌、摩托罗拉等。

性能专家的通常职责是，理解可能会降低软硬件平台性能的每个潜在瓶颈。反应迟缓或性能不佳通常被视为质量问题，所以性能专家的职责就是在构建软件过程中辅助软件工程师和软件设计师使软件产品达到良好的性能水平。

现今世界，随着多层软件体系结构及多种语言编程成为软件开发的主导形式，性能专家的工作变得比 10 年前更加困难。展望未来，10 年之后，性能专家的工作可能变得比现在更加困难。

9.6.13 软件质量保证

- 任务范围 =10 000 功能点
- 潜在缺陷 =5.50
- 缺陷预防影响 =-15%
- 缺陷去除影响 =40%

“质量保证”（QA）通用职位的历史比软件行业还要悠久，它已被工程公司使用近一百年了。在软件世界里，“软件质量保证”（SQA）职位也已存在了 50 多年。今天，在大多数大型软件公司，软件质量专家平均占所有软件雇员的 2% ~ 6%。诸如 IBM 和 Lockheed 等高科技公司比诸如保险和通用制造业等非高科技公司雇用了更多的软件质量保证人员。

当前，只有很小比例的 SQA 人员获得了一个或多个软件质量保证协会的认证。

软件质量保证的职责因公司而异，但通常包括以下这些核心活动：确保使用和遵从有关国际组织和企业的质量标准、度量缺陷去除效率、度量程序控制图的循环复杂度、讲授软件质量课程以及评估或预测软件质量水平。

一些非常成熟的公司（比如 IBM）有质量保证研究职位，研究者可以开发新的或改进的质量控制方法。这些质量保证研究小组的工作成果包括正式审查、功能点度量指标、自动化配置控制工具、净室开发方法以及联合应用设计（JAD）等方法。

软件质量保证职位虽然已存在 50 多年且软件质量保证人员人数数以千计，可为什么当今的软件质量并没有比 1979 年时好多少呢？

原因之一是，在很多公司里质量保证（QA）扮演着咨询角色，但是他们的建议并没有很好地得到遵循。在一些公司里，比如 IBM，在交付一项产品给客户之前，必须获得质量保证团队的正式批准。如果质量保证团队认为缺乏足够的质量控制方法，产品将无法交付。这将是一个非常严重的商业问题。

实际上，很少有项目被阻止交付。但是理论上，如果质量不佳，SQA 有权叫停该软件的交付。这对软件开发团队是一个很强的激励措施，以促使其追求最先进的质量控制方法。

所以，软件质量保证的主要职责是，在低劣质量将导致延迟交付的大背景下，确保最先进的软件质量度量、方法和工具用于软件质量控制。

9.6.14 Web 设计师

- ☐ 任务范围 = 10 000 功能点
- ☐ 潜在缺陷 = 4.00
- ☐ 缺陷预防影响 = -15%
- ☐ 缺陷去除影响 = 12%

软件 Web 设计是一个相当崭新的职业，比几乎任何其他职业增长都快。Web 设计的快速增长，得益于软件公司及其他商业公司将其业务迁移到 Web 上去，使 Web 成为其主要的市场和信息渠道。

在软件质量方面，Web 设计师的职责仍在不断演变，而且还将随着 Web 站点走向虚拟现实和 3-D 表现技术而继续变化。截至 2009 年，Web 设计师的某些职责包括，确保所有接口相当直观，且所有链接都真正工作。

不幸的是，由于黑客行为的指数级增长、数据偷窃和拒绝服务攻击，Web 质量和 Web 安全目前相互交错。Web 站点的有效质量必须包括有效的安全措施，但很多 Web 设计专家对完全行之有效的安全方法知之甚少。

9.6.15 软件需求分析师

- ☐ 任务范围 = 10 000 功能点
- ☐ 潜在缺陷 = 4.00
- ☐ 缺陷预防影响 = -20%
- ☐ 缺陷去除影响 = 15%

需求分析师的工作与系统分析师及业务分析师的工作有所重叠。那些专攻需求分析的人员也了解诸如质量功能展开 (QFD)、联合应用设计 (JAD)、需求审查等主题以及至少半打的需求表现方法, 比如用例、用户故事及其他一些方法。

当前开发的大多数“新”应用软件只不过是那些遗留应用软件的替代品而已, 因此需求分析师也应该熟悉数据挖掘方面的知识和技能。实际上, 开始遗留应用替代者的需求分析的最好入手之处就是挖掘旧遗留应用软件中隐藏在代码里的商业规则和算法。对遗留应用进行数据挖掘很有必要, 因为通常原始规格说明书要么完全丢失, 要么年久失修。

在软件质量方面, 需求分析师的职责是, 确保“毒性”需求在进入设计或编码阶段之前被去除掉。被频繁引用的“千年虫问题”就是毒性需求的一个很好例子。

由于需求阶段之后度量到的需求增长率在每月 1% ~ 3% 之间, 需求分析师的另一项质量责任就是, 确保使用原型法、用户参与、JAD 和其他方法, 以最大限度地减少计划外的需求变更。

需求分析师也应该是负责评审和批准需求变更的需求变更委员会成员, 或者对该委员会提供支持。

9.6.16 软件测试员

- ☐ 任务范围 = 10 000 功能点
- ☐ 潜在缺陷 = 3.00
- ☐ 缺陷预防影响 = -15%
- ☐ 缺陷去除影响 = 50%

软件测试是高度专业化的职业之一。众多经验数据证明, 在软件测试方面, 软件测试专家完全可以超越通才。

但并不是每种测试都要由测试专家来执行。例如, 单元测试几乎总是由开发者自己负责执行。不过, 那些集成整个开发团队工作的测试形式 (包括新功能测试、回归测试和系统测试), 则需要测试大型应用的测试专家。

就软件质量而言, 测试专家的职责是, 确保测试覆盖率达到 99%、测试用例自身不含错误、测试库得以有效维护、清除那些增加成本却没有任何价值的重复测试用例。

虽然不是对当前测试用例开发人员的要求, 但如果测试专家能够度量缺陷去除效率水平, 并努力把平均测试效率从当前的平均大约 35% 提高到至少 75%, 那将对提高软件质量非常有用。

测试专家也应该是新测试技术 (如自动化测试) 的先行者。测试软件之前先运行静态分析工具也会使软件质量有所提高。

9.6.17 功能点专家

- ☐ 任务范围 = 5000 功能点
- ☐ 潜在缺陷 = 4.00
- ☐ 缺陷预防影响 = -10%

□ 缺陷去除影响 = 10%

因为功能点指标是归一化质量数据和创建有效质量信息基准的最好选择，所以功能点专家正快速成为成功软件质量改进计划的一部分。

然而，由于传统手工功能点计数速度太慢且花费不菲，因而其无法作为标准质量控制方法使用。一个经过认证的功能点专家的平均计数速度只有每天大约 400 个功能点。这就解释了对于大于 10 000 个功能点的大型应用软件为什么几乎从来没有使用过功能点分析方法。

所幸的是，人们已经开发出了新的功能点分析方法，允许比以前至少提早 6 个月计算功能点。这些类似方法以超过每分钟 10 000 个功能点的速度进行计算，这使使用功能点方法进行早期质量评估、质量度量以及质量基准预测成为可能。

在软件质量方面，功能点专家的职责是，足够快和尽可能早地创建有用的软件规模信息，以服务于风险分析、软件质量预测和质量度量。

9.6.18 技术文档作家

□ 任务范围 = 2000 功能点

□ 潜在缺陷 = 1.00

□ 缺陷预防影响 = -10%

□ 缺陷去除影响 = 10%

良好的写作能力是人类世界相当稀缺的一种技能。因此，好的软件技术手册也相当稀少。各种质量问题在软件手册中也很常见，包括充满歧义、信息丢失、组织结构混乱以及数据错误等。

当前，已经有很多可用的自动化工具可以用来分析一篇文章的可读性，比如 FOG 指数^①和 Flesch 指数^②方法。但是这些工具很少用于软件手册的检查。将这些文档编辑工具用于软件用户文档的正式审查将会对提高软件文档质量很有帮助。

实际上，另一种已在 IBM 使用的方法是选取那些具有最高用户评价得分的用户文档作为软件用户文档的范例，在以后的用户手册编写中参考并遵循这些范例。

就软件质量而言，技术文档作家的职责是，确保文档中的事实数据完整正确，用户手册易于阅读、便于理解。

9.6.19 软件维护专家

□ 任务范围 = 1500 功能点

□ 潜在缺陷 = 3.50

□ 缺陷预防影响 = -30%

① FOG Index，又称为 Gunning Fog Index、Gunning Readability Formula，是度量一篇文章的阅读难易程度的可读性测试工具。详细信息请参见 http://en.wikipedia.org/wiki/Gunning-Fog_Index。——译者注。

② 又称 Flesch/Flesch-Kincaid 可读性测试。与 Fog 指数类似，表明当阅读英文文章一个段落时，读者理解的困难程度。详细信息请参见 http://en.wikipedia.org/wiki/Flesch-Kincaid_readability_test。——译者注

□ 缺陷去除影响 = 20%

在遗留应用软件功能增强和软件缺陷修复方面,软件维护编程工作已成为过去 20 多年软件行业的主导性活动。这并不值得惊讶,因为在任何超过 50 年历史的行业里,工作在现存产品修复上的人都比工作在新开发产品上的人多。

随着经济衰退的加深和延续,美国汽车行业就提供了一个非常痛苦的明证:汽车制造业萎缩的速度比两极冰原融化的速度还快,而汽车修理业却在快速增长。

老旧的遗留应用软件存在诸多问题,包括程序结构不良、部分代码僵死、存在易错模块以及代码注释不足或缺失。

随着经济衰退的持续,很多公司都在考虑如何延伸遗留应用程序的使用年限。事实上,即使面临经济衰退,遗留应用的改造和数据挖掘业务也都还在增长。

在软件质量方面,软件维护程序员的主要职责是,增强遗留应用软件的质量。实现此目的可行方法包括使用自动化工具对遗留应用进行完全改造、度量和降低遗留应用的代码复杂度、去除僵死代码、改进注释、易错模块的识别与外科手术式去除、将代码从孤儿语言(如 MUMPS 或 Coral 语言)转换为现代编程语言(比如 Java 或 Ruby 语言)以及消除遗留应用中的安全缺陷。

9.6.20 审查会议主持人

□ 任务范围 = 1000 功能点

□ 潜在缺陷 = 4.50

□ 缺陷预防影响 = -25%

□ 缺陷去除影响 = 35%

软件审查有许多标准角色,包括会议主持人、会议记录者、审查者和被审查者。主持人是审查会议成功的关键角色。主持人的任务包括确保会议按计划讨论既定事项,最大限度地减少破坏性事件,确保审查会议按时开始并按时结束。

在软件质量方面,审查会议主持人的主要职责包括确保即将审查的材料及时交付到审查者手中以作预习,确保审查者和其他人员及时到场,保持审查团队精力集中在缺陷识别(与修复)上并干预潜在的争论与纠纷。

审查记录者也扮演了至关重要的角色,因为记录者撰写会议记录,填写审查所识别的所有错误和缺陷的问题报告。这项工作并没有看起来那么容易,因为审查过程中可能会有一些辩论,以讨论某个具体问题是一个缺陷还是一个潜在的功能增强。

9.6.21 软件专家总结

软件工程文献中并没有很好地涵盖软件行业的所有主题。鉴于目前软件行业存在着 115 种与软件有关的各种专家,这种状况有些令人惊讶。

当涉及软件质量时,由前述缺陷预防和缺陷去除的分析可见,某些形式的专业化分工确实会促使软件质量得到提高。使软件质量价值增加最多的关键专家包括风险分析师、六西格玛专家、质量保证人员、审查会议主持人、软件维护专家和专业软件测试人员。

同时，比如业务分析师、企业架构师、软件架构师、软件估算专家和功能点专家等其他许多专家，也能够使软件质量得到不小的提高。

9.7 软件质量的经济价值

目前存在的软件工程文献中并没有很好地涉及软件质量的经济价值这一主题。出现这种情况有诸多原因。其中的主要原因是软件工程领域的软件质量度量方法相当糟糕。很多成本因素比如无偿加班经常被忽略不计。此外，软件成本数据中频繁出现疏忽和遗漏，比如项目管理成本的遗漏、兼职专家（如技术文档作家）的遗漏等。实际上，只有编码工作的成本有相当不错的可用数据。其他任何工作，比如需求、设计、审查、测试、质量保证、项目办公室以及文档方面的数据，往往少报或干脆忽略不报。

正如本书其他部分指出的那样，软件工程文献过于依赖含糊不清、不可预测的质量定义，比如“软件产品符合用户需求”或者满足一系列“特性”。这些不科学的质量定义使软件质量经济价值的研究进展缓慢。

其他两个无效经济度量指标的使用也影响了软件质量经济价值的研究，即：平均缺陷成本和代码行。正如本章早先讨论过的，平均缺陷成本对质量不利，达到最低缺陷成本的软件却常常是千疮百孔。代码行对高级编程语言不利，它掩盖了高级编程语言在软件质量和生产力研究方面的应有价值。

这一部分，笔者将尝试用 8 个研究案例来展示软件质量的经济价值。由于软件质量的经济价值与应用规模紧密相关，所以使用 4 个离散的规模数量级来举例说明：100 个功能点、1000 个功能点、10 000 个功能点和 100 000 个功能点。

100 个功能点量级的应用软件通常是大型系统的一个小功能模块而不是一个独立的应用软件。但是，这个量级也是较大型应用软件原型最常见的规模范围。可能有些小型独立应用软件的规模也在这个规模量级，比如货币转换器或者手持设备（如 iPhone）上的小程序等之类的小型应用软件。

1000 个功能点量级的应用软件通常是独立应用软件，比如燃油喷射控制系统、原子表控制软件、诸如 Java 等编程语言的编译器以及 COCOMO 之类的软件估算工具等。

10 000 个功能点量级的应用软件通常是业务控制各个方面的重要系统，比如保险理赔处理、机动车登记、儿童支持应用软件等软件系统。

100 000 个功能点量级的应用软件通常是大型国际电话交换系统之类的重大系统、Vista 或者 IBM MVS 之类的操作系统、微软 Office 之类相互关联协作的应用套件。一些 ERP 应用软件也属此类规模的软件，甚至有可能达到 300 000 个功能点。同样，诸如全球军事指挥与控制系统（WWMCCS）等大型国防应用软件也可能高达 100 000 个功能点。

为减少可变因素数量，所有 8 个示例均假定以 C 语言编写代码，且每个功能点包含大约 125 个代码语句。

由于所有 8 个应用示例均假定使用相同编程语言编写，故生产力和质量可以用无失真的代码行指标来表示。注意，将代码行指标用于不同编程语言之间的比较是无效的。

对于上述每个规模量级，会分别用两个案例加以说明：平均质量和优良质量。假设平

均质量的案例中使用瀑布开发方法、CMMI 1 级、常规测试方法以及在缺陷预防方面没有任何特别之处的开发方法。

而优良质量的案例中则假设至少使用了 CMMI 3 级、正式审查、静态分析、严格的开发方法比如团队软件过程 (TSP) 及需求收集的原型法和联合应用设计 (JAD) 等方法。

(有些读者可能会奇怪该研究案例为什么没有使用敏捷开发方法。其主要原因是目前还没有达到 10 000 和 100 000 个功能点规模量级的敏捷应用软件。敏捷方法主要应用于 1000 功能点量级的小型应用软件开发。)

尽管所有研究案例均来自真实应用软件项目, 为使计算保持一致, 这里使用了许多简化假设, 包括以下几个关键点:

- ❑ 所有成本数据均基于每人每月 10 000 美元的完全负担成本。每个工作人员每月工作 132 小时, 相当于每小时 75.75 美元。
- ❑ 假设每月工作 22 天, 每天 8 小时。无偿加班和有偿加班均未考虑。
- ❑ 潜在缺陷数目是已发现的 5 大类别缺陷的总数: 需求缺陷、设计缺陷、代码缺陷、文档缺陷以及不良修复或者缺陷修复中意外引入的次生缺陷。
- ❑ 不包括需求蔓延。8 个案例研究的规模反映了最终交付给客户的应用软件规模。
- ❑ 不包括软件重用。假设所有 8 个案例均重用了约 15% 的遗留旧代码。但为了简化, 假设重用代码和其他材料中的潜在缺陷等于新开发材料中的潜在缺陷。大量认证可重用材料的使用将极大提高所有 8 个研究案例的质量和生产力, 对超过 10 000 个功能点的大型系统尤其如此。
- ❑ 也不考虑不良修复注入。大约 7% 的缺陷修复会意外引入新错误, 但既然引起这些错误的起源开发活动都没有发现它们, 不良修复注入的数学计算将非常复杂。
- ❑ 假设软件维护阶段的第一年发现了随软件交付给客户的所有潜伏缺陷。现实中, 许多缺陷潜伏多年后才会出现, 但这里的例子只考虑维护阶段第一年发现的缺陷。
- ❑ 软件维护数据只考虑缺陷修复。为强调软件质量价值, 功能增强和添加新功能均排除在外。
- ❑ 软件维护缺陷修复率基于每人每月修复 12 个缺陷这一平均值。实际项目中, 其取值范围可能从每月低于 4 个到超过 20 个缺陷不等。
- ❑ 软件应用团队的人员数量基于美国各类软件人员的平均任务范围而计算, 大约每人 150 个功能点。也就是说, 以功能点计算的应用规模大小除以技术工人加上项目管理人员的总员工编制, 其结果接近每人 150 个功能点。这个数值包括了软件工程师和质量保证、技术文档作家和测试人员等专家。
- ❑ “平均”质量案例的进度估计值提升至功能点规模的 0.4 次幂。这个经验规则较好地提供了一种从需求开始到交付应用之时以月度计算的进度粗略估计。
- ❑ “优良”质量案例的进度估计值提升至功能点规模的 0.36 次幂。这个指数在面向对象软件开发方法和质量控制比较严格的软件开发实践活动中效果很好。对于敏捷项目也是个不错的选择, 但除了那些超过 10 000 个功能点的敏捷项目, 因为缺乏该类项目的度量数据, 笔者不确定该指数是否适合此类项目。

- 本节数据使用国际功能点用户组（IFPUG）4.2 版计数规则所定义的功能点指标来表示。其他的功能性指标，如 COSMIC 功能点、工程功能点、Mark II 功能点，所产生的结果与这里所示数据有所不同。
- 本节的源代码数据使用逻辑语句数表示，不使用物理代码行数。根据代码计数是以物理行数还是逻辑行数的不同，表面上看起来相同的代码数量规模可能实际上会有高达 5 倍的差异。

进行这些简化假设的目的是为了最大限度地减少 8 个研究案例中的无关变化，以一致的方式表示每个研究案例的数据。由于所有这些假设在实际软件项目中均有变化，请读者使用自己的本地数据或者来自诸如国际软件基准组织（ISBSG）的基准数据等替代值进行尝试。

这些简化假设是为了使结果一致，但在实际工作中这些假设可能差别很大。

9.7.1 微型应用软件（100 个功能点）的质量价值

该数量级范围的微型应用软件通常具有较低的潜在缺陷和相当高的缺陷去除效率水平。这是因为，此类小应用通常可由一个人开发，因而不存在不同个体或团队所开发功能之间的接口问题。表 9-24 显示了 100 个功能点微型应用的质量价值。

表 9-24 100 个功能点^①规模应用软件的质量价值

	平均质量	优良质量	差异
平均每功能点缺陷	3.50	1.50	-2.00
潜在缺陷	350	150	-200.00
缺陷去除效率	94.00%	99.00%	5.00%
去除的缺陷	329	149	-181
交付的缺陷	21	2	-20
发布前平均缺陷成本	379 美元	455 美元	76 美元
发布后平均缺陷成本	1 061 美元	1 288 美元	227 美元
开发周期（自然月）	6	5	-1
开发人员数目	1	1	0
开发工作量（人月）	6	5	-1
开发成本	63 096 美元	52 481 美元	-10 615 美元
每人月功能点	15.85	19.05	3.21
每人月代码行	1 981	2 382	401
维护人员数量	1	1	0
维护工作量（人月）	2	0	-1.63
维护成本（第一年）	17 500 美元	1 250 美元	-16 250 美元
总工作量	8	5	-3
总成本	80 596 美元	53 731 美元	-26 865 美元
平均每人总成本	40 298 美元	26 865 美元	-13 432 美元
平均功能点总成本	805.96 美元	537.31 美元	-269 美元
平均代码行总成本	6.45 美元	4.30 美元	-2.15 美元
平均缺陷成本	720 美元	871 美元	152 美元

① 100 功能点 = 12 500 条 C 语言语句。

需要注意的是,平均缺陷成本随着质量的提高而升高却不是下降。这种现象扭曲了经济分析,正如随后例子中所看到的,平均缺陷成本往往随着应用规模的增大而下降。因为大型应用的缺陷要比小型应用多得多。

应用软件原型和本数量级范围内的应用软件对个体开发者的技能水平非常敏感,主要原因是个体开发者一个人几乎承担了所有开发工作。一个给定功能到底需要编写多少代码,不同个体开发者的代码数量差异可能会高达5倍。就生产力和软件质量水平而言,有可能达到6倍。因此,在此类项目上使用平均值需要格外小心,对于由一个人完成大部分开发工作的应用软件,其平均值尤其不可靠。

9.7.2 中小型应用软件(1000个功能点)的质量价值

对于1000个功能点规模数量级的小型应用软件,软件质量开始变得非常重要,而且某种程度上该类应用软件也比大型系统更容易实现高质量。该量级规模的应用软件,需要的开发团队较小,诸如敏捷开发等现代开发方法往往占主导地位,而团队软件过程(TSP)及Rational统一过程(RUP)等质量控制极其严格的开发方法则在系统应用和嵌入式软件中更为常见。表9-25展示了1000个功能点量级中小型应用的质量价值。

表 9-25 1000 个功能点^①应用软件的质量价值

	平均质量	优良质量	差异
平均功能点缺陷	4.50	2.50	-2.00
潜在缺陷	4500	2500	-2 000
缺陷去除效率	93.00%	97.00%	4.00%
去除的缺陷	4185	2425	-1760
交付的缺陷	315	75	-240.00
发布前平均缺陷成本	341 美元	417 美元	76 美元
发布后平均缺陷成本	909 美元	1136 美元	227 美元
开发周期(自然月)	16	12	-4
开发人员数	7	7	0.00
开发工作量(人月)	106	80	-26
开发成本	1 056 595 美元	801 510 美元	255 086 美元
每人月功能点	9.46	12.48	3.01
每人月代码行	1183	1 560	376.51
维护人员数量	2	2	0
维护工作量(人月)	26	6	-20.00
维护成本(第一年)	262 500 美元	62 500 美元	200 000 美元
总工作量	132	86	-46
总成本	1 319 095 美元	864 010 美元	455 086 美元
平均每人总成本	158 291 美元	103 681 美元	-54 610 美元
平均功能点总成本	1 319.10 美元	864.01 美元	-455 美元
平均代码行总成本	10.55 美元	6.91 美元	-3.64 美元
每缺陷平均成本	625 美元	776 美元	152 美元

① 1000 功能点=125 000 条 C 语言语句。

表 9-25 “优良质量”列数据显示了该研究案例中软件项目的良好软件质量结果，产生这些良好软件质量的原因是项目中使用了需求、设计和代码的正式审查，减少了缺陷的产生，进而缩短了软件测试周期。能够增加质量价值的其他方法还包括团队软件过程（TSP）、测试之前的静态分析以及较高的 CMMI 等级等。

在 1000 个功能点数量级规模上，很多软件方法都相当有效。例如，敏捷开发和极限编程都能够产生很不错的软件质量，Rational 统一过程（RUP）和团队软件过程（TSP）亦同样如此。

9.7.3 大型应用软件（10 000 个功能点）的质量价值

10 000 个功能点数量级的应用软件都是非常重要的系统软件，要求密切关注质量控制、变更控制和公司治理。实际上，如果没有认真谨慎的软件质量和变更控制，这个规模数量级范围内的项目，失败或取消的项目能占全部软件项目的 35% 以上。

注意，即使实施了成熟的质量控制措施，随着应用软件规模的增加，潜在缺陷也将会快速增长，而缺陷去除效率水平则有所下降。这是由于需求和设计的书面工作量呈指数级增长，经常导致只能审查一部分可交付物而不是审查 100% 全部可交付物。对于大型系统，测试覆盖率也会有所下降，需要的测试用例数目快速上升，但经常仍然无法跟上软件复杂度的增长。表 9-26 说明了当软件规模上升到 10 000 个功能点数量级时的软件质量价值。

随着软件规模的增加，由于卓越的软件质量（缺陷较少）而带来的成本节约也随之增加。一般规则是，软件应用规模越大，其卓越的软件质量所能带来的经济价值也越多。同样原则还适用于变更控制，因为需求蔓延数量也会随着软件规模的增加而上升。

对于大型应用，已被证明能够有效改善该类软件质量的可用方法开始减少。例如，敏捷方法很难应用于此规模数量级的应用软件，即使勉强应用，其结果也并不理想。对于大型应用软件，质量控制极其严格的开发方法，比如 Rational 统一过程（RUP）或者团队软件过程（TSP）能够产生最好的质量效果，目前还具有最大数量的经验数据。

表 9-26 10 000 个功能点^①规模数量级应用软件的质量价值

	平均质量	优良质量	差异
平均功能点缺陷	6.00	3.50	-2.50
潜在缺陷	60 000	35 000	-25 000
缺陷去除效率	84.00%	96.00%	12.00%
去除的缺陷	50 400	33 600	-16 800
交付的缺陷	9600	1400	-8200
发布前平均缺陷成本	341 美元	417 美元	76 美元
发布后平均缺陷成本	833 美元	1 061 美元	227 美元
开发周期（自然月）	40	28	-12
开发人员数	67	67	0
开发工作量（人月）	2 654	1836	-818
开发成本	26 540 478 美元	18 361 525 美元	-8 178 953 美元

(续)

	平均质量	优良质量	差异
每人月功能点	3.77	5.45	1.68
每人月代码行	471	681	209.79
维护人员数量	17	17	0
维护工作量(人月)	800	117	-683.33
维护成本(第一年)	8 000 000 美元	1 166 667 美元	-6 833 333 美元
总工作量	3454	1953	-1501
总成本	34 540 478 美元	19 528 191 美元	-15 012 287 美元
平均每人总成本	414 486 美元	234 338 美元	-180 147 美元
平均功能点总成本	3 454.05 美元	1 952.82 美元	-1501.23 美元
平均代码行总成本	27.63 美元	15.62 美元	-12.01 美元
平均缺陷成本	587 美元	739 美元	152 美元

① 10 000 功能点=1 250 000 条 C 语言语句。

9.7.4 超大型应用软件(100 000 个功能点)的质量价值

100 000 个功能点数量级的应用软件是现代商业中最昂贵的开发工程。这些超大型系统经常危险重重,因为有很多软件项目失败了,而几乎所有的软件项目都存在预算超支、开发进度落后的状况。

如果没有卓越的软件质量控制措施,成功完成 100 000 个功能点的应用软件开发的可能性仅为 20%,而在规划的时间和预算内完成项目的可能性几乎为零。

即使有卓越的质量控制和非凡的变更控制,大量 100 000 个功能点规模数量级的应用软件也常常是成本昂贵、麻烦不断。表 9-27 说明了这种超大规模应用软件的两种情况。

表 9-27 100 000 个功能点^①应用的质量价值

	平均质量	优良质量	差异
平均功能点缺陷	7.00	4.00	-3.00
潜在缺陷	700 000	400 000	-300 000
缺陷去除效率	81.00%	94.00%	13.00%
去除的缺陷	567 000	376 000	-191 000
交付的缺陷	133 000	24 000	-109 000
发布前平均缺陷成本	303 美元	379 美元	76 美元
发布后平均缺陷成本	758 美元	985 美元	227 美元
开发周期(自然月)	100	63	-37
开发人员数	667	667	0
开发工作量(人月)	66 667	42 064	-24 603
开发成本	666 666 667 美元	420 638 230 美元	-246 028 437 美元
每人月功能点	1.50	2.38	0.88
每人月代码行	188	297	109.67

(续)

	平均质量	优良质量	差异
维护人员数量	167	167	0
维护工作量(人月)	11 083	2 000	-9 083
维护成本(第一年)	110 833 333 美元	20 000 000 美元	-90 833 333 美元
总工作量	77 750	44 064	-33 686
总成本	777 500 000 美元	440 638 230 美元	-336 861 770 美元
平均每人总成本	933 000 美元	528 766 美元	-404 234 美元
平均功能点总成本	7 775.00 美元	4 406.38 美元	-3 368.62 美元
平均代码行总成本	62.20 美元	352.51 美元	290.31 美元
每缺陷平均成本	530 美元	682 美元	152 美元

① 100 000 功能点=12 500 000 条 C 语言语句。

为什么大型应用软件的潜在缺陷如此之高？为什么超大型应用的缺陷去除效率水平会降低？对于这些问题，可能的原因很多。首先，这类大规模应用软件的需求变更数量非常庞大，常常超出大多数公司的软件质量控制能力。

其次，随着应用软件规模的增加，书面工作量往往也急剧增加。这些书面工作阻碍了需求和设计的正式审查等软件质量活动。结果，超大型应用软件往往只能在主要可交付物上执行部分审查而不是 100% 全部审查。

再次，IBM 在 20 世纪 70 年代曾计算出来，为达到 90% 的代码测试覆盖率，测试用例数量将随着应用软件规模的增加而呈指数级增长。实际上，要完全测试一个 100 000 个功能点的超大型应用软件，需要的测试用例数目趋近于无穷大。结果是，即使静态分析和正式审查工作保持不变，测试效率随着规模增加而急剧下降。

预测测试用例总数的一个有效经验法则是，测试用例总数是以功能点计算的应用软件规模的 1.2 次幂。由此可见，所需测试用例数量的增长非常迅速，大多数公司无法同步跟上测试用例增加的要求，因而测试覆盖率出现下降。对于 100 000 个功能点的超大型应用软件，自动化静态分析方法仍然有效，正式审查也同样有效，但对关键可交付物只进行部分审查而不再是 100% 全部审查已变得司空见惯，因为项目中的书面工作量也随着应用软件规模的增加而呈指数级增长。

9.7.5 软件质量的投资回报 (ROI)

正如前面已经提到过的，卓越的软件质量所带来的经济价值随着应用软件规模的增长而增长。表 9-28 分别计算出了 100 个功能点、1 000 个功能点、10 000 个功能点和 100 000 个功能点的应用软件良好质量案例的大致投资回报。

为易于计算和理解，这里同样做了简化。基本假设是，每一位软件团队成员需要 5 天培训以使团队成员快速掌握软件审查和团队软件过程 (TSP) 方法。然后将培训天数乘以每位员工的平均每小时成本 75.75 美元得到总培训费用。

项目节省的资金总数 (包括因卓越的软件质量而节省下来的开发和维护费用) 除以上述

总培训费用所得最终结果就是大概的投资回报。表 9-28 展示了计算所得的投资回报数据。

表 9-28 软件质量的投资回报

功能点规模	100	1000	10 000	100 000
培训小时数	80	560	5360	53 360
培训成本	6060 美元	42 420 美元	406 020 美元	4 042 020 美元
高质量节约	26 865 美元	455 086 美元	15 012 287 美元	336 861 770 美元
投资回报	4.43 美元	10.73 美元	36.97 美元	83.34 美元

ROI 数据反映了项目节省下来的总费用除以使团队成员快速掌握软件质量技术所需要培训的费用所得的结果。

在实际软件项目中，上述假设条件可能有很大出入，还需要考虑很多其他项目因素。即便如此，由于卓越的软件质量可以使开发周期更短、开发费用和维护费用更低，因而高水平的软件质量具有非常可观的投资回报。

软件项目中可能还需要就其他软件质量主题对软件工程师和经理们进行培训，以及还需要考虑更多其他项目成本（比如实现能力成熟度模型（CMM）较高等级的成本）。虽然经常可以在实际项目中观察到卓越的软件质量带来的项目成本节约，但根据当地财务规则处理培训和过程改进费用的方式不用，精确的投资回报计算结果亦有所不同。

如果将项目取消或重大项目预算超支等项目风险降低也包含到投资回报（ROI）计算中，则最终 ROI 值将会更高。

其他技术（如大量的认证重用材料）也将对软件质量和生产力产生有益影响。然而，写作本书的 2009 年，可用的认证可重用材料来源非常有限。未经认证的可重用材料会使软件质量变得非常危险，甚至对项目质量有害而无害。

9.8 总结

尽管软件行业花费在寻找和修复软件缺陷上的资金比其他任何软件活动都多，但软件工程文献中所涵盖的软件质量主题的讨论却仍然含混不清、水平参差不齐。

关于软件质量和软件测试的书籍多如牛毛，但包含缺陷数量、测试用例数量、测试覆盖率或者缺陷去除活动相关成本等量化数据的书籍却少之又少。

更糟糕的是，很多软件质量文献只是简单引述“平均缺陷成本在整个开发期间逐步上升”这一“都市传说”，却很少能够真正认识到这是因为忽视了固定成本所致。

软件质量确实具有极高的经济价值，且这种价值会随着软件应用规模的增加而增加。实际上，没有卓越的质量控制，即使“完成”一个大型软件应用也是几乎不可能的。而没有过硬的质量控制，按时并在预算范围内完成软件项目则更是天方夜谭。

参考文献

Beck, Kent. *Test-Driven Development*. Boston, MA: Addison Wesley, 2002. (中文版《测试驱动开发》，崔凯

- 译, 中国电力出版社 2004 年 4 月出版。)
- Chelf, Ben and Raoul Jetley. *Diagnosing Medical Device Software Defects Using Static Analysis*. San Francisco, CA: Coverity Technical Report, 2008.
- Chess, Brian and Jacob West. *Secure Programming with Static Analysis*. Boston, MA: Addison Wesley, 2007.(中文版《安全编程代码静态分析》, 董启雄、韩平、程永敬译, 机械工业出版社 2008 年 3 月出版。)
- Cohen, Lou. *Quality Function Deployment—How to Make QFD Work for You*. Upper Saddle River, NJ: Prentice Hall, 1995.
- Crosby, Philip B. *Quality is Free*. New York, NY: New American Library, Mentor Books, 1979.(最新中文版《质量免费》, 杨钢、林海译, 山西出版集团山西教育出版社 2011 年 6 月出版。)
- Everett, Gerald D. and Raymond McLeod. *Software Testing*. Hoboken, NJ: John Wiley & Sons, 2007. (中文版《软件测试跨越整个软件开发生命周期》, 郭耀译, 清华大学出版社 2008 年 9 月出版。)
- Gack, Gary. Applying Six Sigma to Software Implementation Projects. <http://software.isixsigma.com/library/content/c040915b.asp>.
- Gilb, Tom and Dorothy Graham. *Software Inspections*. Reading, MA: Addison Wesley, 1993.
- Hallowell, David L. Six Sigma Software Metrics, Part 1. <http://software.isixsigma.com/library/content/c03910a.asp>.
- International Organization for Standards. ISO 9000 / ISO 14000. <http://www.iso.org/iso/en/iso9000-14000/index.html>.
- Jones, Capers. *Software Quality—Analysis and Guidelines for Success*. Boston, MA: International Thomson Computer Press, 1997.
- Kan, Stephen H. *Metrics and Models in Software Quality Engineering*, Second Edition. Boston, MA: Addison Wesley Longman, 2003. (中文版《软件质量工程——度量与模型》(第二版), 吴明晖、应晶译, 电子工业出版社 2004 年 7 月出版。)
- Land, Susan K., Douglas B. Smith, John Z. Walz. *Practical Support for Lean Six Sigma Software Process Definition: Using IEEE Software Engineering Standards*. Los Alamitos, CA: Wiley-IEEE Computer Society Press, 2008.
- Mosley, Daniel J. *The Handbook of MIS Application Software Testing*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall, 1993.
- Myers, Glenford. *The Art of Software Testing*. New York, NY: John Wiley & Sons, 1979. (最新中文版《软件测试的艺术》(原书第 3 版), 张晓明、黄琳译, 机械工业出版社 2012 年 4 月出版。)
- Nandyal, Raghav. *Making Sense of Software Quality Assurance*. New Delhi: Tata McGraw-Hill Publishing, 2007.
- Radice, Ronald A. *High Quality Low Cost Software Inspections*. Andover, MA: Paradoxicon Publishing, 2002.
- Wieggers, Karl E. *Peer Reviews in Software—A Practical Guide*. Boston, MA: Addison Wesley Longman, 2002. (中文版《软件同级评审》, 沈备军、宿为民译, 机械工业出版社 2003 年 6 月出版。)

推荐阅读



算法导论（原书第3版）

2006、2007 CSDN、《程序员》杂志评选的
十大IT好书之一 算法中的经典权威之作

编译原理（原书第2版）

编译领域无可替代的经典著作，被广大计算机
专业人士誉为“龙书”

设计模式：可复用面向对象软件的基础
经典教材 权威之作

软件工程（原书第8版）

最受欢迎的软件工程指南

数据库系统概念（原书第5版）

数据库系统方面的经典教材，被美誉为
“帆船书”

软件工程：实践者研究方法（原书第6版）

全球上百所大学和学院采用 最受欢迎的软件工
程指南